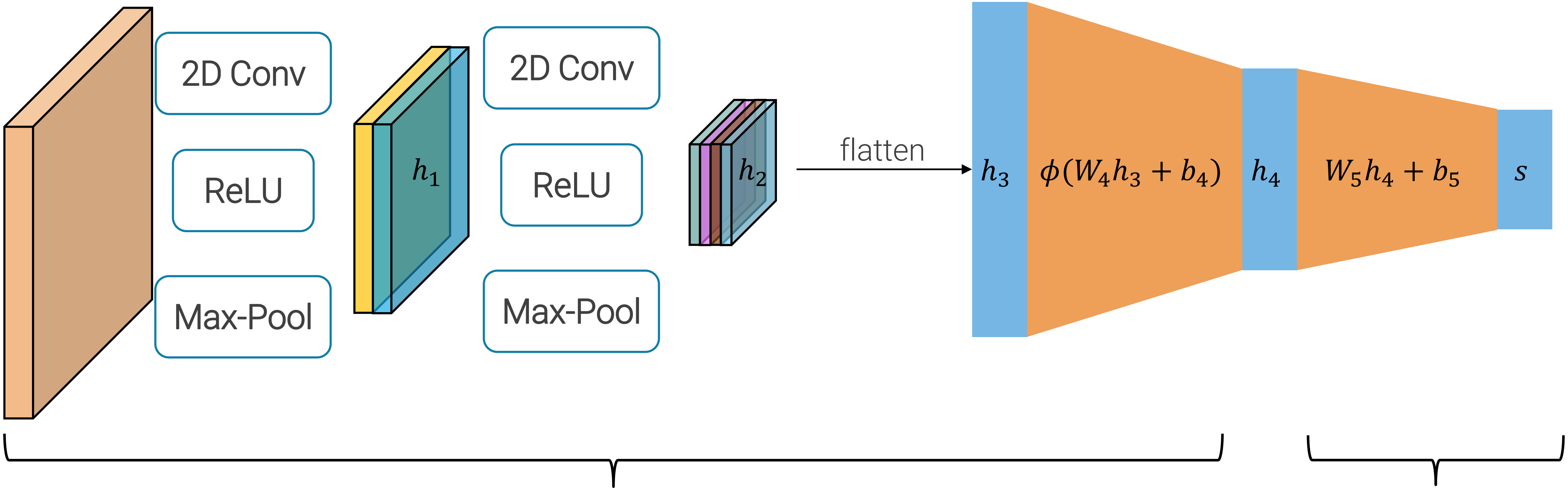
Lecture 5
Successful Architectures

IMAGE PROCESSING AND COMPUTER VISION – PART 2

SAMUELE SALTI

Convolutional Neural Networks

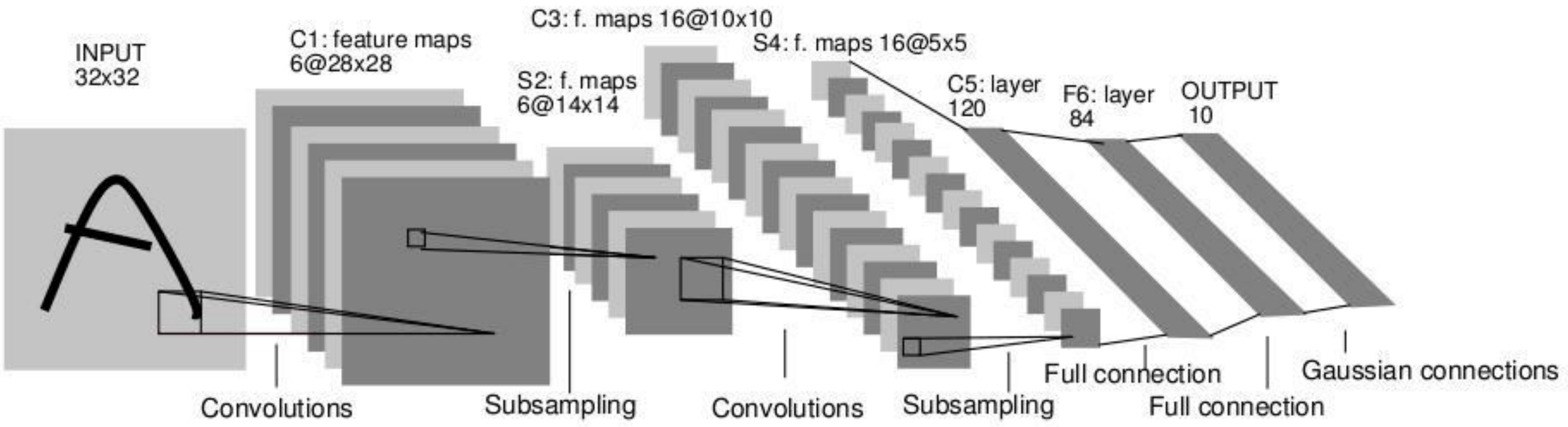


N convolutional+pooling layers followed by M fully connected layers
This is also called the feature extractor
(with max-pool, ReLU can be the last operation in a block)

The final fully connected layer is also called the classifier

Example: LeNet5

num_ch grows inside the network while we reduce the spatial dimension ---> stil valid nowadays

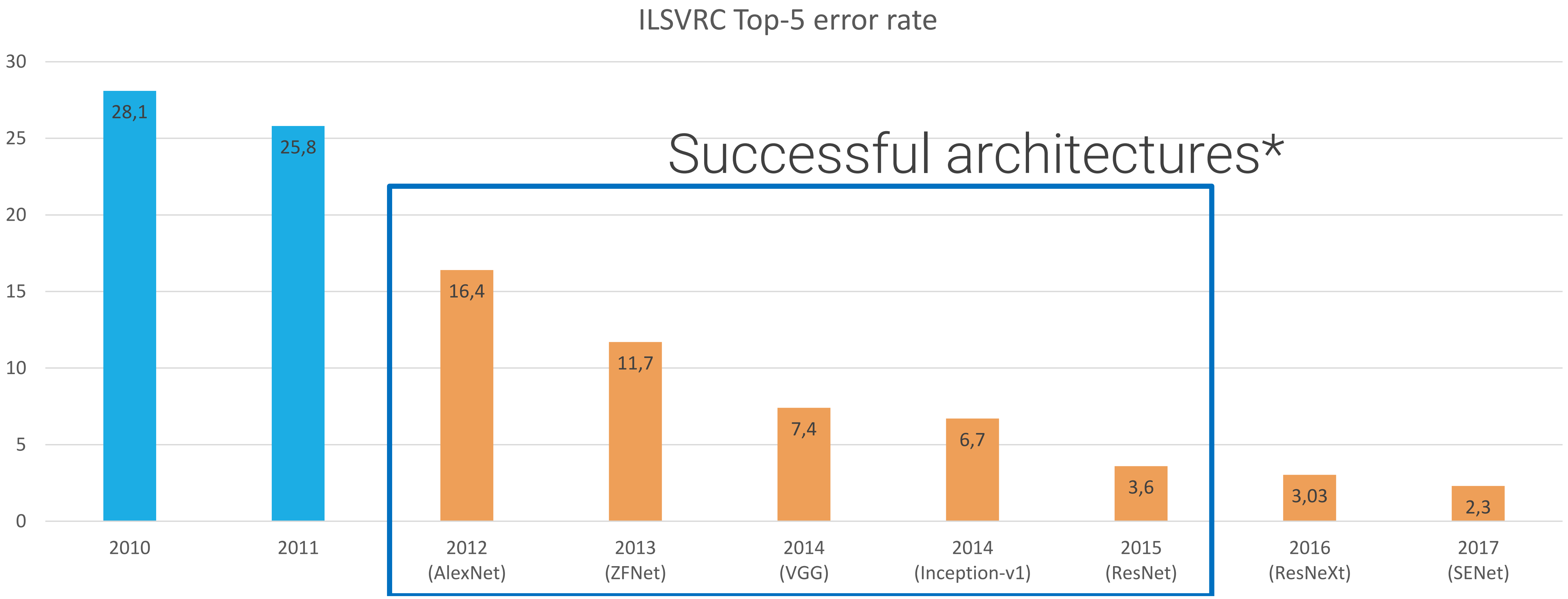


- As depth increases, number of channels increase, and spatial dimension decreases
- Average pooling
- 5x5 convolutional kernels, no padding
- Sigmoid or tanh non-linearities with carefully selected amplitudes
- Multiple fully connected layers + RBF classifier
- No residual connections
- No (batch) normalization

It may seems like in 90s they were doing neural networks as we do but in practice they were different (in red)

Lecun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. "Gradient-based learning applied to document recognition", Proceedings of the IEEE. 1998.

ILSVRC error rate evolution

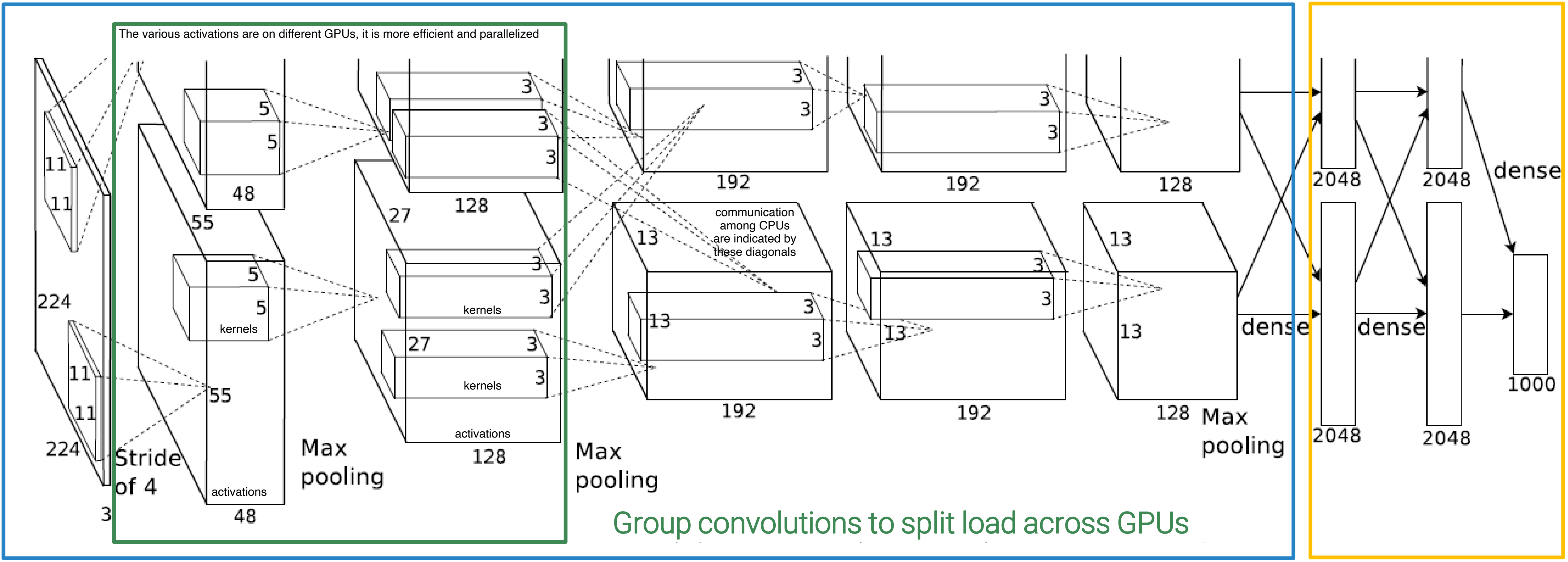


*Results based on ensembles and, sometimes, heavy test-time augmentation

AlexNet

5 convolutional layers (conv+relu) optionally followed by max-pooling

3 fc layers



Group convolutions to split load across GPUs

AlexNet

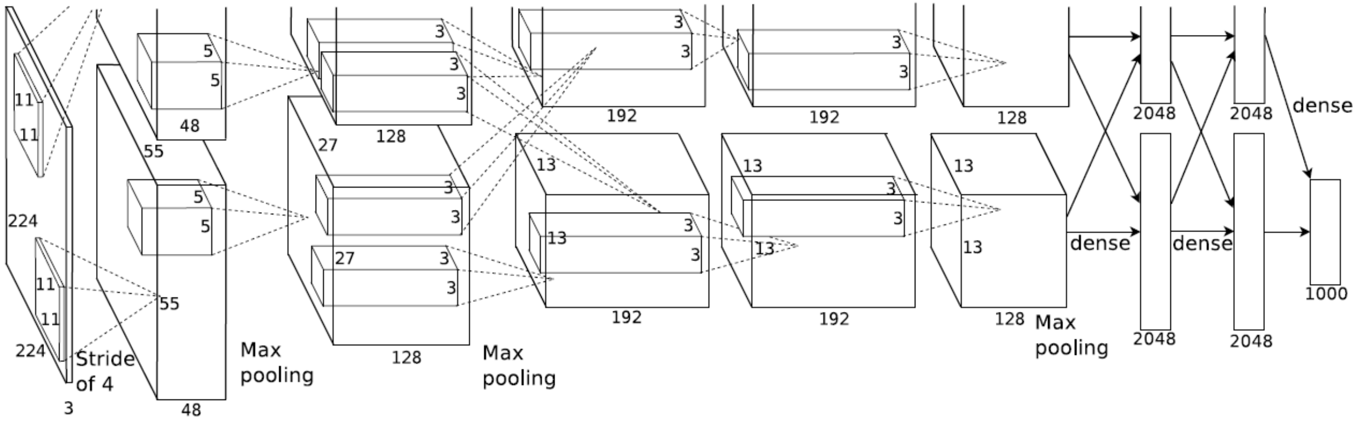
Won ILSVRC 2012.

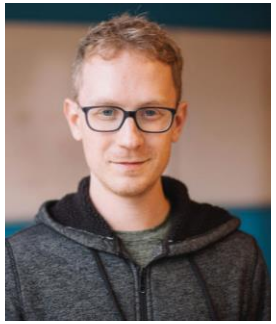
Was trained on two GTX580 GPUs.

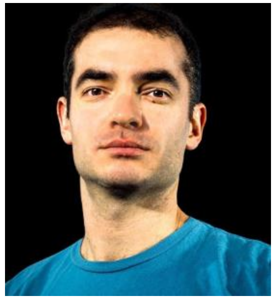
Used local response normalization (LRN) in some layers, not used in subsequent architectures.

Took between five and six days to train

“All our experiments suggest that our results can be improved simply by waiting for faster GPUs and bigger datasets to become available.”

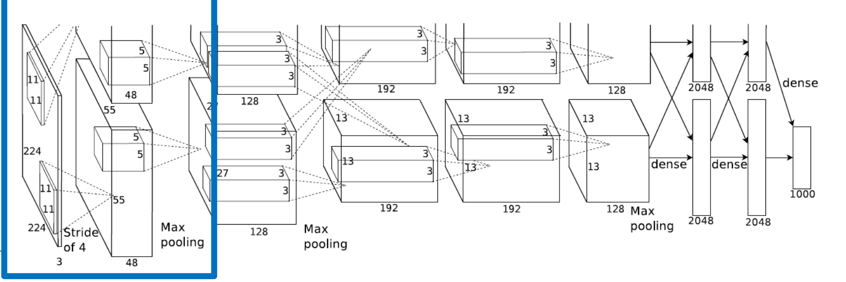






Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, NeurIPS 2012

Architecture breakdown



Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0

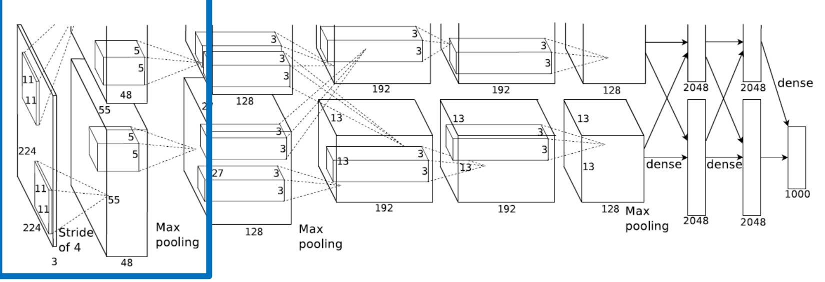
Input elements = $W_{in} \times H_{in} \times C_{in}$
=227*227*3
=154,587

AlexNet used mini-batch size 128,
activations are usually floating point number, i.e. 4 bytes for each element.

Total memory in MB for a mini-batch is
128* 154,587 *4/1024/1024
= 75.5

Minibatch:	128

Architecture breakdown



Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0							

First layer is a 96x3x11x11 convolutional layer, with stride 4.
We will see again this stem layer at the beginning of convnets, i.e. a conv layer that performs a fast reduction in the spatial size of the activations, mainly to reduce memory and computational cost, but also to rapidly increase the receptive field.

Output channel = # kernels = 96

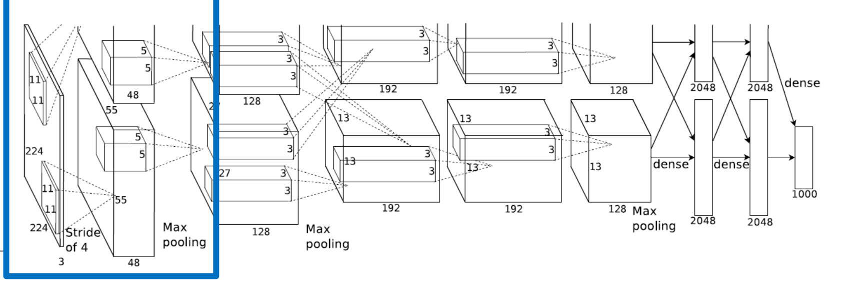
$$\text{Activation H/W} = \left\lfloor \frac{(W_{in} - W_K + 2P)}{S} \right\rfloor + 1$$

$$= (227-11+0) / 4 + 1$$

$$=216/4+1=55$$

Minibatch:	128

Architecture breakdown



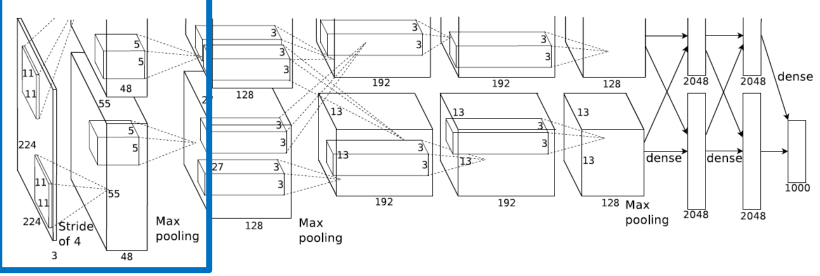
Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400				

params = $(\boldsymbol{W_K} \times \boldsymbol{H_K} \times \boldsymbol{C_{in}} + \mathbf{1}) \times \boldsymbol{C_{out}}$
=(11*11*3+1)*96
=34,944

flops = #activations $\times (\boldsymbol{W_K} \times \boldsymbol{H_K} \times \boldsymbol{C_{in}}) \times \mathbf{2}$
=290,400 * (11*11*3)*2
=27 Gflops

Minibatch:	128

Architecture breakdown



Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3		

Given activation size and #parameters, how much memory will we need at training time?

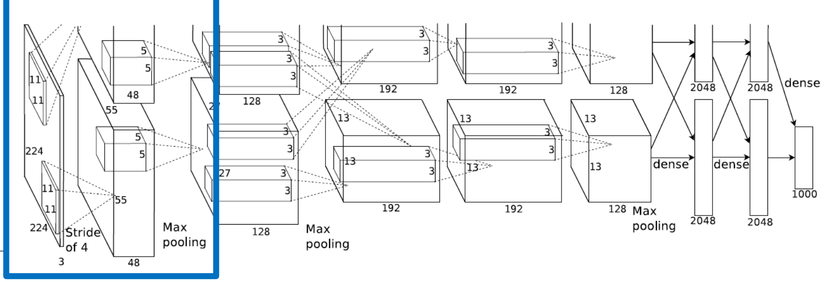
- We need to store all intermediate activations, in order to compute the gradient of the loss with respect to everyone of its entries, i.e. another tensor of the same size
- We will have as many activations (and gradients) as there are input images in our mini-batch

- For every parameter, we will need to store its value and the gradient of the loss with respect to it
- If we use advanced optimizers like momentum, we will have a velocity term for each parameter, if using Adam first and second order moments for each parameter, etc..

Hard to have a precise estimate of memory requirements, but we can get approximate values by considering twice the activation size and 3-4 times the #params. This is usually a lower bound on the actual requirements.

Minibatch:	128

Architecture breakdown



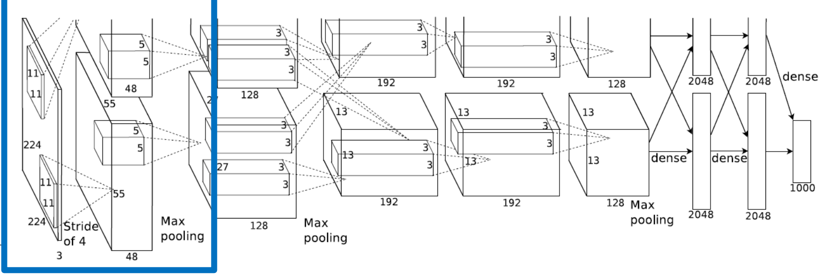
Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3		

Total memory for activations for conv1 is then
2 * 128 * #activations * 4 / 1024/1024=
283.6 MB

Total memory for parameters for conv1 is then
3*#params * 4 / 1024/1024=
0.4 MB

Minibatch:	128

Architecture breakdown



Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0							

All pooling layers are "overlapping" pooling layers: they have size 3x3, but stride 2. Reduces the top-1 and top-5 errors in their experiments compared to 2x2 with stride 2.

output channels = # input channels = 96

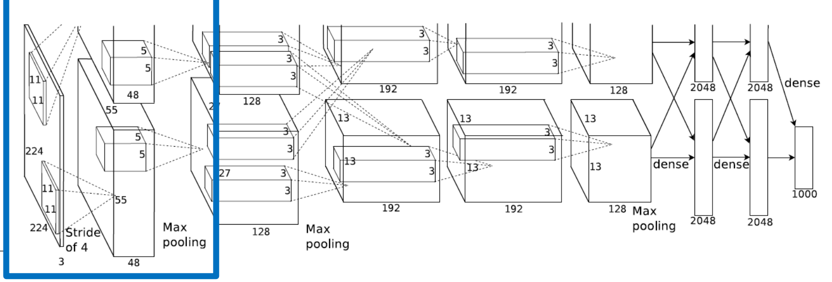
$$\text{Activation H/W} = \left\lfloor \frac{(W_{in} - W_K + 2P)}{S} \right\rfloor + 1$$

$$= (55-3+0) / 2 + 1$$

$$=52/2+1=27$$

Minibatch:	128

Architecture breakdown



Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0	27	96	69984				

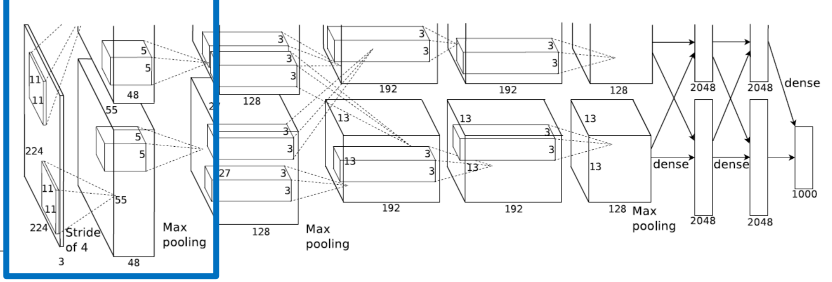
params = 0 -> size of params in memory = 0 MB

flops = #activations $\times$ $(\boldsymbol{W_K} \times \boldsymbol{H_K})$
=69,984 * 3 * 3
= 629,856 = 0.6 Mflops

Size of activations for pool1 is then
2 * 128 * #activations * 4 / 1024/1024=
68.3 MB

Minibatch:	128

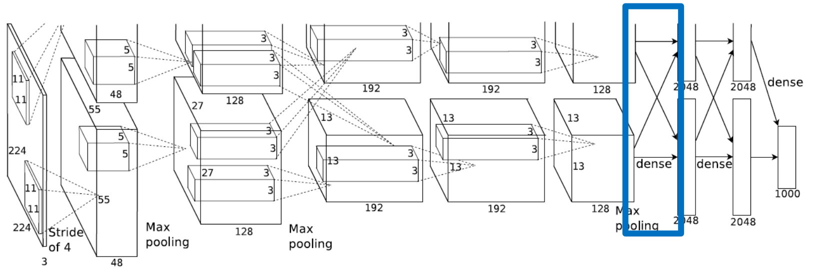
Architecture breakdown



Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0	27	96	69984	0	80,6	68,3	0,0

Minibatch:	128

Architecture breakdown

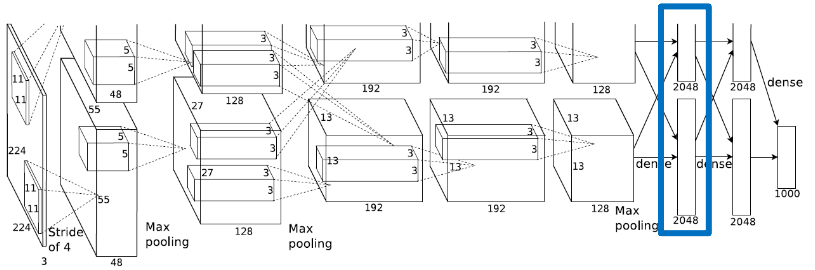


Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0	27	96	69984	0	80,6	68,3	0,0
conv2	256	5	1	2	27	256	186624	615	114661,8	182,3	7,0
pool2	1	3	2	0	13	256	43264	0	49,8	42,3	0,0
conv3	384	3	1	1	13	384	64896	885	38277,2	63,4	10,1
conv4	384	3	1	1	13	384	64896	1327	57415,8	63,4	15,2
conv5	256	3	1	1	13	256	43264	885	38277,2	42,3	10,1
pool3	1	3	2	0	6	256	9216	0	10,6	9,0	0,0
flatten	0	0	0	0	1	9216	9216				

Flatten layer to throw away the spatial structure and prepare for FC layers. No computation, so no parameters. Not even memory consumption, it is just a view over the same area of memory which is interpreted as a $C_{out} \times 1$ vector, where $C_{out} = C_{in} \times W_{in} \times \mathrm{H}_{in}$

Minibatch:	128

Architecture breakdown

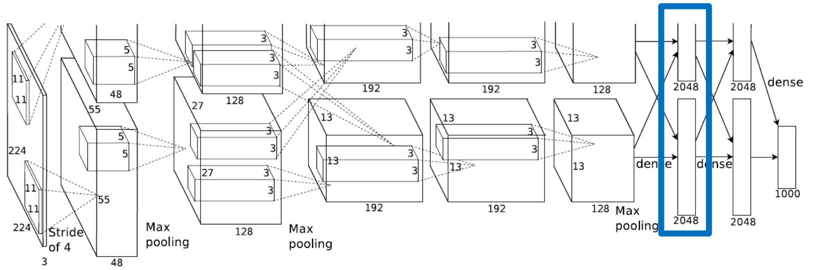


Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0	27	96	69984	0	80,6	68,3	0,0
conv2	256	5	1	2	27	256	186624	615	114661,8	182,3	7,0
pool2	1	3	2	0	13	256	43264	0	49,8	42,3	0,0
conv3	384	3	1	1	13	384	64896	885	38277,2	63,4	10,1
conv4	384	3	1	1	13	384	64896	1327	57415,8	63,4	15,2
conv5	256	3	1	1	13	256	43264	885	38277,2	42,3	10,1
pool3	1	3	2	0	6	256	9216	0	10,6	9,0	0,0
flatten	0	0	0	0	1	9216	9216	0	0,0	0,0	0,0
fc6	4096	1	1	0							

\# output channels = # kernels = 4096 , no spatial dimensions -> memory of actvs = $2*128*4096*4/1024^2$= 4 MB

Minibatch:	128

Architecture breakdown

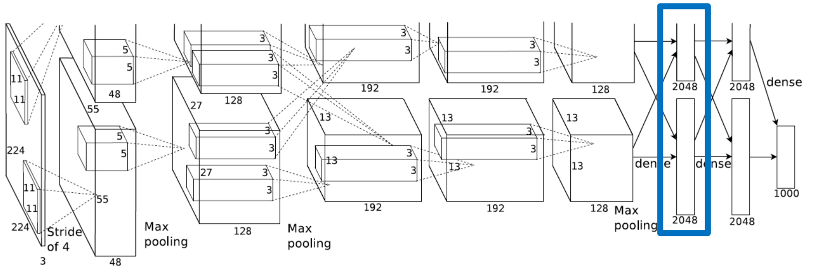


Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0	27	96	69984	0	80,6	68,3	0,0
conv2	256	5	1	2	27	256	186624	615	114661,8	182,3	7,0
pool2	1	3	2	0	13	256	43264	0	49,8	42,3	0,0
conv3	384	3	1	1	13	384	64896	885	38277,2	63,4	10,1
conv4	384	3	1	1	13	384	64896	1327	57415,8	63,4	15,2
conv5	256	3	1	1	13	256	43264	885	38277,2	42,3	10,1
pool3	1	3	2	0	6	256	9216	0	10,6	9,0	0,0
flatten	0	0	0	0	1	9216	9216	0	0,0	0,0	0,0
fc6	4096	1	1	0	1	4096	4096			4,0	

params = $C_{out} \times C_{in} + C_{out}$ = 37,752,832 -> memory for params = 3 * #params *4 / 1024^2 = 432,1 MB

Minibatch:	128

Architecture breakdown

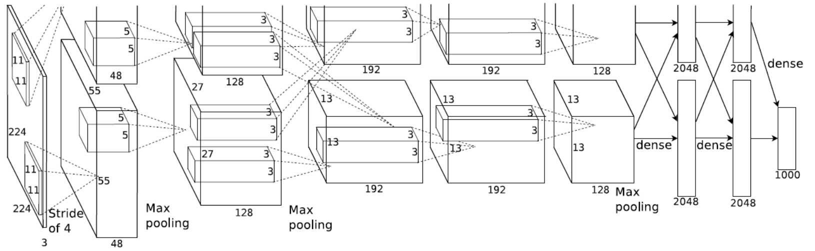


Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0	27	96	69984	0	80,6	68,3	0,0
conv2	256	5	1	2	27	256	186624	615	114661,8	182,3	7,0
pool2	1	3	2	0	13	256	43264	0	49,8	42,3	0,0
conv3	384	3	1	1	13	384	64896	885	38277,2	63,4	10,1
conv4	384	3	1	1	13	384	64896	1327	57415,8	63,4	15,2
conv5	256	3	1	1	13	256	43264	885	38277,2	42,3	10,1
pool3	1	3	2	0	6	256	9216	0	10,6	9,0	0,0
flatten	0	0	0	0	1	9216	9216	0	0,0	0,0	0,0
fc6	4096	1	1	0	1	4096	4096	37758		4,0	432,0

flops = $\#minibatch \times 2 \times C_{in} \times \#\ activations$ = 128*2*9,126*4,096 = 9.569 Gflops

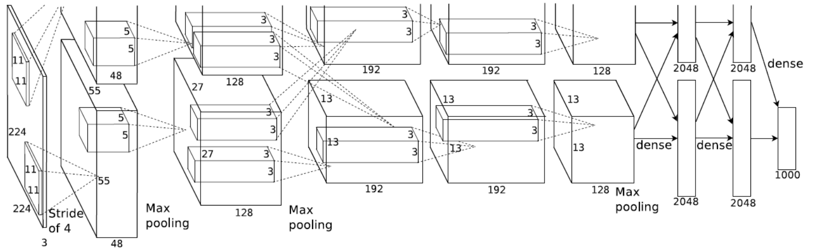
Minibatch:	128

Architecture breakdown



Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0	27	96	69984	0	80,6	68,3	0,0
conv2	256	5	1	2	27	256	186624	615	114661,8	182,3	7,0
pool2	1	3	2	0	13	256	43264	0	49,8	42,3	0,0
conv3	384	3	1	1	13	384	64896	885	38277,2	63,4	10,1
conv4	384	3	1	1	13	384	64896	1327	57415,8	63,4	15,2
conv5	256	3	1	1	13	256	43264	885	38277,2	42,3	10,1
pool3	1	3	2	0	6	256	9216	0	10,6	9,0	0,0
flatten	0	0	0	0	1	9216	9216	0	0,0	0,0	0,0
fc6	4096	1	1	0	1	4096	4096	37758	9663,7	4,0	432,0
fc7	4096	1	1	0	1	4096	4096	16781	4295,0	4,0	192,0
fc8	1000	1	1	0	1	1000	1000	4097	1048,6	1,0	46,9
					Minibatch:	128	Totals:	62378	290851	1.406	714

Architecture breakdown



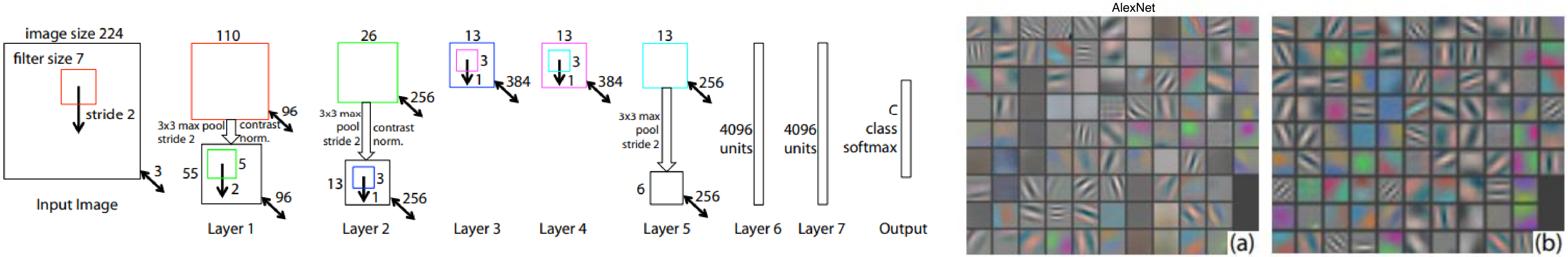
Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0	27	96	69984	0	80,6	68,3	0,0
conv2								615	114661,8	182,3	7,0
pool2								0	49,8	42,3	0,0
conv3								885	38277,2	63,4	10,1
conv4								1327	57415,8	63,4	15,2
conv5								885	38277,2	42,3	10,1
pool3								0	10,6	9,0	0,0
flatten								0	0,0	0,0	0,0
fc6								37758	9663,7	4,0	432,0
fc7								16781	4295,0	4,0	192,0
fc8	1000	1	1	0	1	1000	1000	4097	1048,6	1,0	46,9
					Minibatch:	128	Totals:	62378	290851	1.406	714

Trends to note:

- Stem layer at the beginning of the network
- Nearly all parameters are in the fully connected layers
- Largest memory consumption from activations due to the first conv layers
- Largest number of flops required by the conv layers
- Activations and parameters have a similar memory footprint at training time
- More than 60 million parameters learned, more than 290 Gflops to process a mini-batch, 2.2 Gflops/image

Not ptimal way

ZFNet / Clarifai: a better AlexNet



First author founded a company, Clarifai, which won ILSVRC 2013 with a modified version of this network.

Tries to reduce the “trail and error” approach to network design, by introducing powerful visualizations (via Deconvnets) for layers other than the first one.

Based on the visualizations and ablation studies, they found out that aggressive stride and large filter size in the first layer results in dead filters and missing frequencies in the first layer filters and aliasing artifacts in the second layer activations

They propose to counteract these problems by using **7×7 convs with stride 2** in the first layer and **stride 2 also in the second 5×5 conv layer.**

Matthew D. Zeiler and Rob Fergus, “Visualizing and Understanding Convolutional Networks”, ECCV 2014

VGG: Deep but regular

Second place in ILSVRC 2014, 7.5% top-5 error

Commit to explore the effectiveness of simple design choices, by allowing only the combination of :

- 3x3 convolutions, S=1, P=1
- 2x2 max-pooling, S=2, P=0
- #channels doubles after each pool

Dropped local response normalization (LRN)

Batch norm not invented yet! Pre-initialization of deeper networks with weights from shallower architectures crucial to let training progress (unless smart initialization strategies are used).

VGG-16 VGG-19

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Karen Simonyan and Andrew Zisserman, “Very Deep Convolutional Networks for Large-scale Image Recognition”, ICLR 2015

Stages

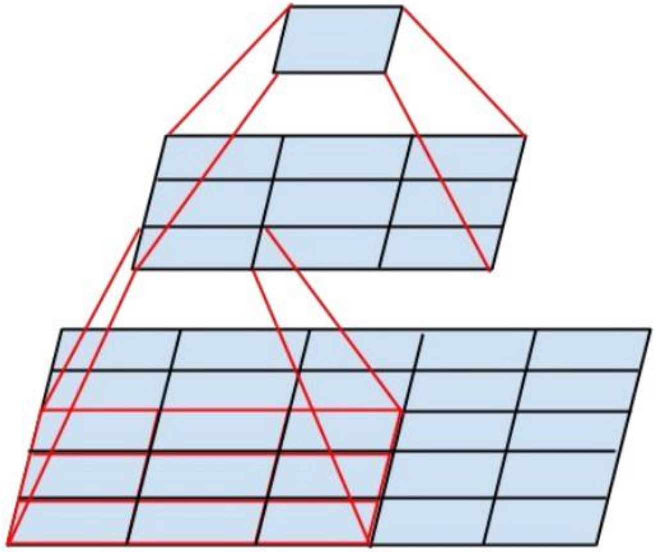
VGG introduces the idea of designing a network as repetitions of **stages**, i.e. a fixed combination of layers that process activations **at the same spatial resolution**.

In VGG, stages are either:

- conv-conv-pool
- conv-conv-conv-pool
- conv-conv-conv-conv-pool

One stage **has same receptive field of larger convolutions** but **requires less params and computation** and introduces more non-linearities.

No free-lunch, though: **memory for activations doubles**



A lot of 7x7/5x5 convolution can be emulated my stacking smaller kernels
In that way we use less params -> less flops

More non linearities —> more expressive nework

Conv layer	Params	Flops	ReLUs	#Activations
$C \times C \times 5 \times 5\ , S = 1, P = 2$	$25C^2 + C$	$50C^2 W_{in} H_{in}$	1	$C \times W_{in} \times H_{in}$
2 stacked $C \times C \times 3 \times 3, S = 1, P = 1$	$18C^2 + 2C$	$36C^2 W_{in} H_{in}$	2	$2 \times C \times W_{in} \times H_{in}$

VGG-16	VGG-19
D 16 weight layers	E 19 weight layers
conv3-64 conv3-64	conv3-64 conv3-64
conv3-128 conv3-128	conv3-128 conv3-128
conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool	maxpool
FC-4096	FC-4096
FC-4096	FC-4096
FC-1000	FC-1000
soft-max	soft-max

VGG-16 summary

Bigger network than AlexNet

138 M params (2.3x AlexNet) again, mostly in fc layers

~4 Tflops (~ 14x), 31 Gflops/img, mainly due to convolutions

~16.5 GB of memory (~ 12x) Memory mostly stores activations (of first conv layers), which are much larger than in AlexNet for the absence of stem layers)

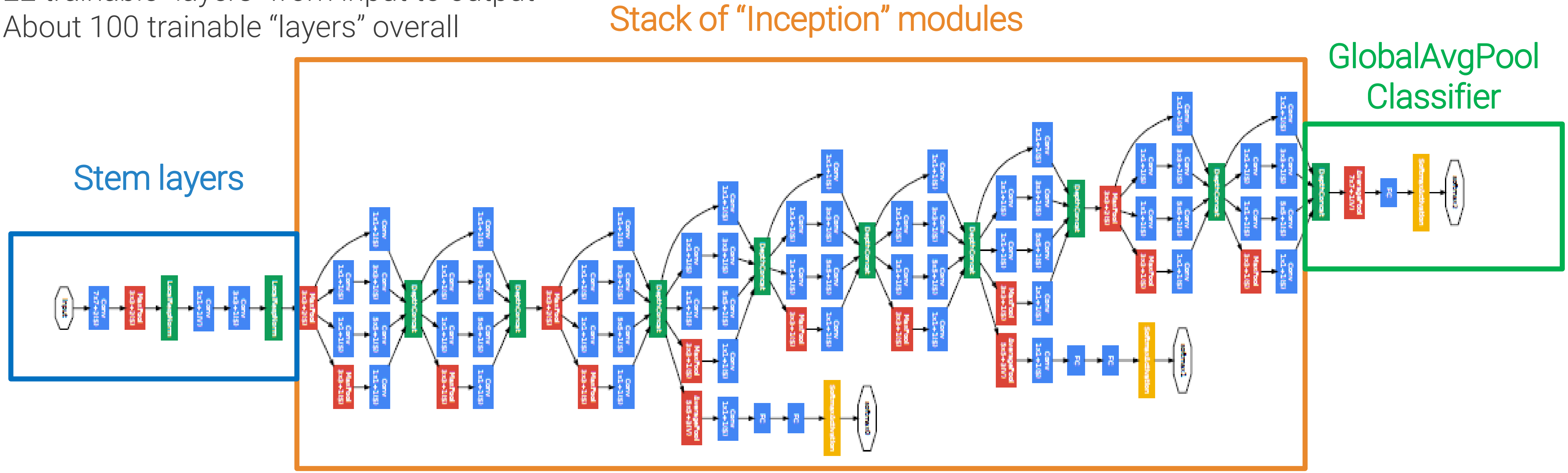
Trained on 4 GPUs with data parallelism for 2-3 weeks

layer	Kernels	K_W/H	S	P	Actv H/W	Actv Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Params memory (MB)
input					224	3	150528	0	-	73,5	0,0
conv1	64	3	1	1	224	64	3211264	2	22196,3	3136,0	0,0
conv2	64	3	1	1	224	64	3211264	37	473520,1	3136,0	0,4
pool1	1	2	2	0	112	64	802816	0	411,0	784,0	0,0
conv3	128	3	1	1	112	128	1605632	74	236760,1	1568,0	0,8
conv4	128	3	1	1	112	128	1605632	148	473520,1	1568,0	1,7
pool2	1	2	2	0	56	128	401408	0	205,5	392,0	0,0
conv5	256	3	1	1	56	256	802816	295	236760,1	784,0	3,4
conv6	256	3	1	1	56	256	802816	590	473520,1	784,0	6,8
conv7	256	3	1	1	56	256	802816	590	473520,1	784,0	6,8
pool3	1	2	2	0	28	256	200704	0	102,8	196,0	0,0
conv8	512	3	1	1	28	512	401408	1180	236760,1	392,0	13,5
conv9	512	3	1	1	28	512	401408	2360	473520,1	392,0	27,0
conv10	512	3	1	1	28	512	401408	2360	473520,1	392,0	27,0
pool4	1	2	2	0	14	512	100352	0	51,4	98,0	0,0
conv11	512	3	1	1	14	512	100352	2360	118380,0	98,0	27,0
conv12	512	3	1	1	14	512	100352	2360	118380,0	98,0	27,0
conv13	512	3	1	1	14	512	100352	2360	118380,0	98,0	27,0
pool5	1	2	2	0	7	512	25088	0	12,8	24,5	0,0
flatten	1	1	1	0	1	25088	25088	0	0,0	0,0	0,0
fc14	4096	1	1	0	1	4096	4096	102786	26306,7	4,0	1176,3
fc15	4096	1	1	0	1	4096	4096	16781	4295,0	4,0	192,0
fc16	1000	1	1	0	1	1000	1000	4100	1048,6	1,0	46,9
					Minibatch:	128	Totals:	138.382	3.961.171	14.733	1.584

Inception v1 (GoogLeNet)

“The main hallmark of this architecture is the improved utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for increasing the depth and width of the network while keeping the computational budget constant.”

22 trainable “layers” from input to output
About 100 trainable “layers” overall



Christian Szegedy et al., “Going deeper with convolutions”, CVPR 2015

Stem layers

	inception 1x1				inception 3x3			Inception 5x5			Maxpool		Activations					Acts	Params
Layer	Ks	H/W	S	P	C_out	1x1	H/W	C_out	1x1	H/W	1x1	H/W	H/W	Channels	#Activations	#params (K)	flops (M)	memory (MB)	memory (MB)
input													224	3	150528	0	-	73,5	0,0
conv1	64	7	2	3									112	64	802816	9	30211,6	784,0	0,1
pool1	1	3	2	1									56	64	200704	0	231,2	196,0	0,0
conv2	64	1	1	0									56	64	200704	4	3288,3	196,0	0,0
conv3	192	3	1	1									56	192	602112	111	88785,0	588,0	1,3
pool2	1	3	2	1									28	192	150528	0	173,4	147,0	0,0

Stem layers aggressively downsamples inputs: from 224 to 28 width/height in 5 layers, which require about 130 G glops, 124 K parameters, and 2 GB of memory

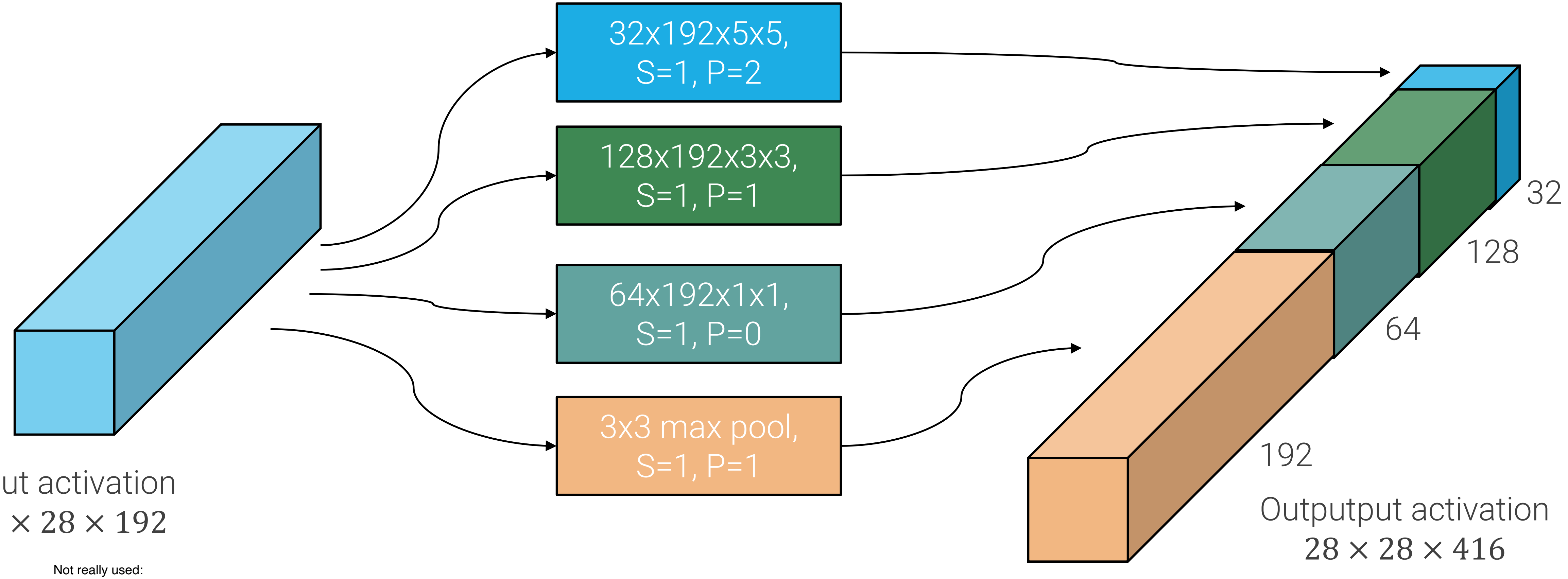
Yet, it brings it down a bit more gently than AlexNet, as per ZFNet lesson

- only uses strides of 2
- largest conv is 7x7

Compare with VGG: to reach 28x28, it uses 10 layers, with more than half of the total flops (2.4 Tflops), more than 1.7 M parameters, and 13 GB of memory.

Naïve Inception module

Instead of make a decision about which kernel to use, we give all these convolutions to SGD and it decide the best ways to use them

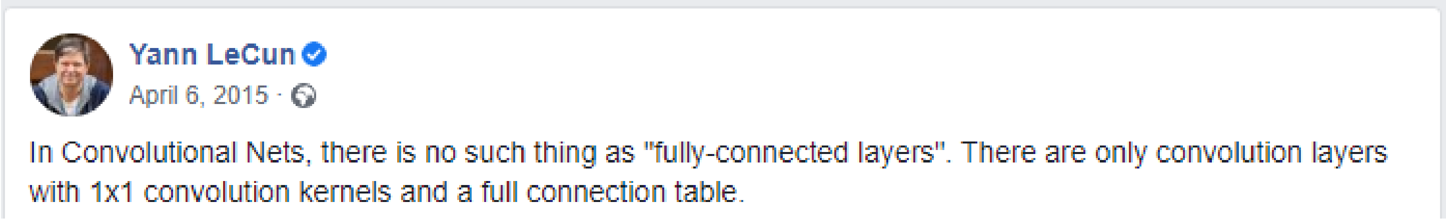


Not really used:

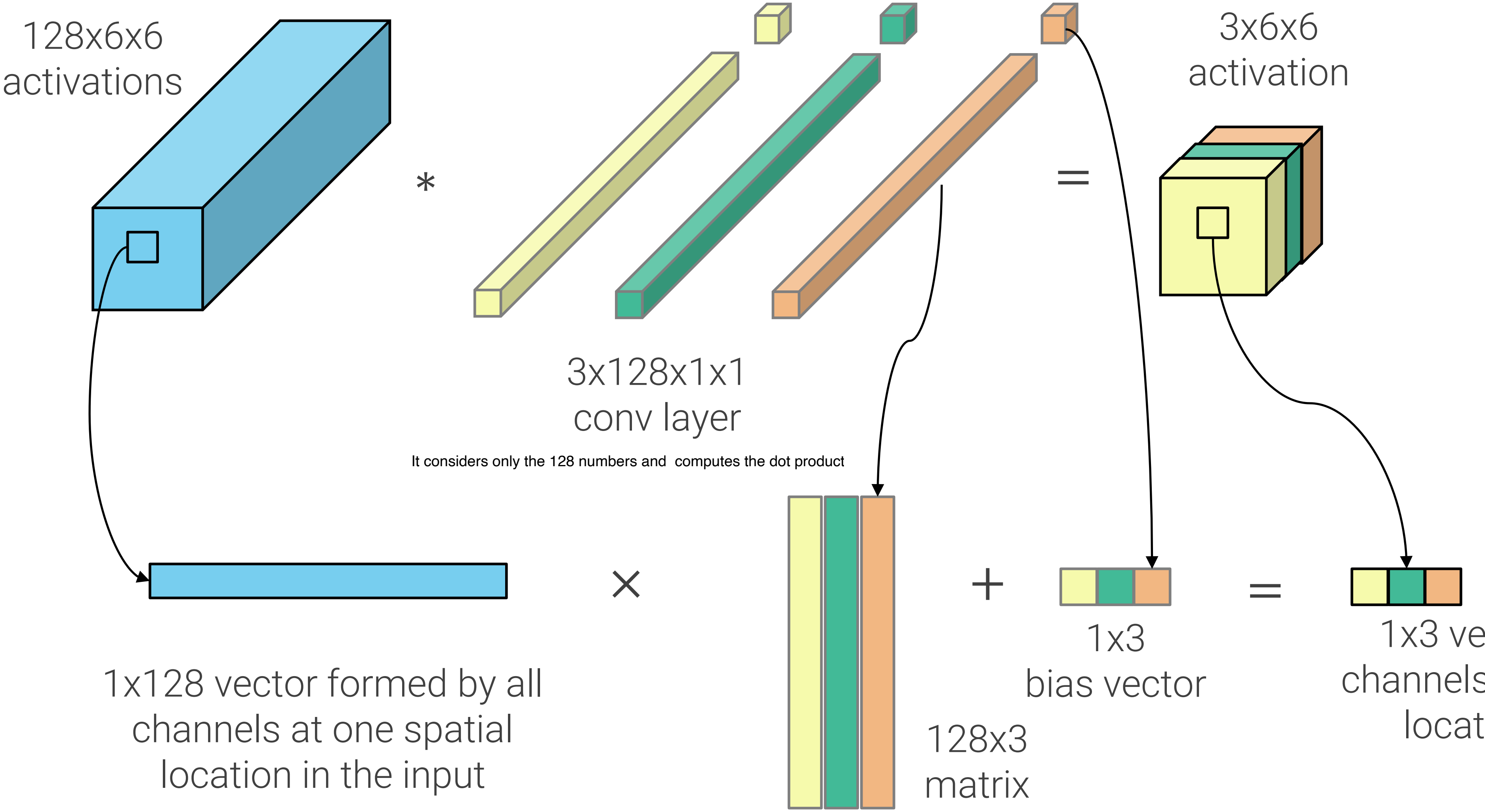
Two main problems:

- Due to max-pool, #channels grows very fast when inception modules are stacked on top of each other
- 5x5 and 3x3 convs on many channels become prohibitively expensive if we stack a lot of them, e.g. here
  conv 5x5 = 2x28x28x32x192x5x5=240 Mflops, conv 3x3 = 2x28x28x128x192x3x3=350 Mflops

1 x 1 convolutions


Yann LeCun
April 6, 2015 ·

In Convolutional Nets, there is no such thing as "fully-connected layers". There are only convolution layers with 1x1 convolution kernels and a full connection table.



1 x 1 convolutions behaves like all other convolutions.

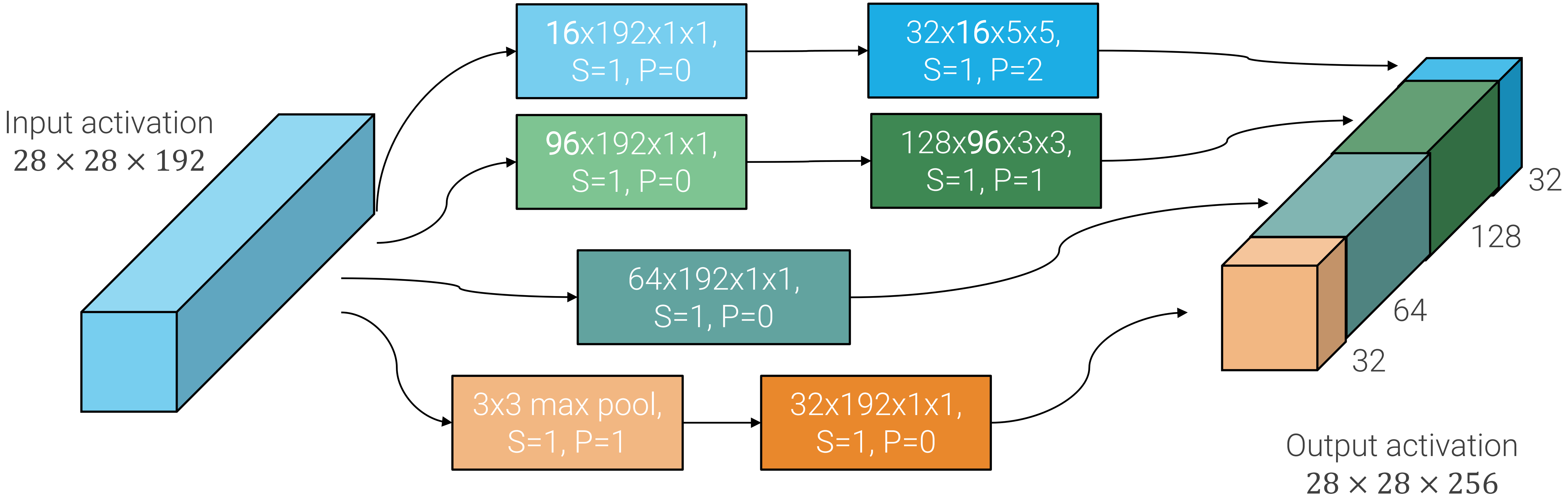
They allows us to change the depth of the activations while preserving the spatial size.

We can also interpret them as applying a linear FC layer at each spatial location.

If we stack several of them we are actually applying an MLP at each location.

Lin et al., "Network in network", ICLR 2014

Inception module



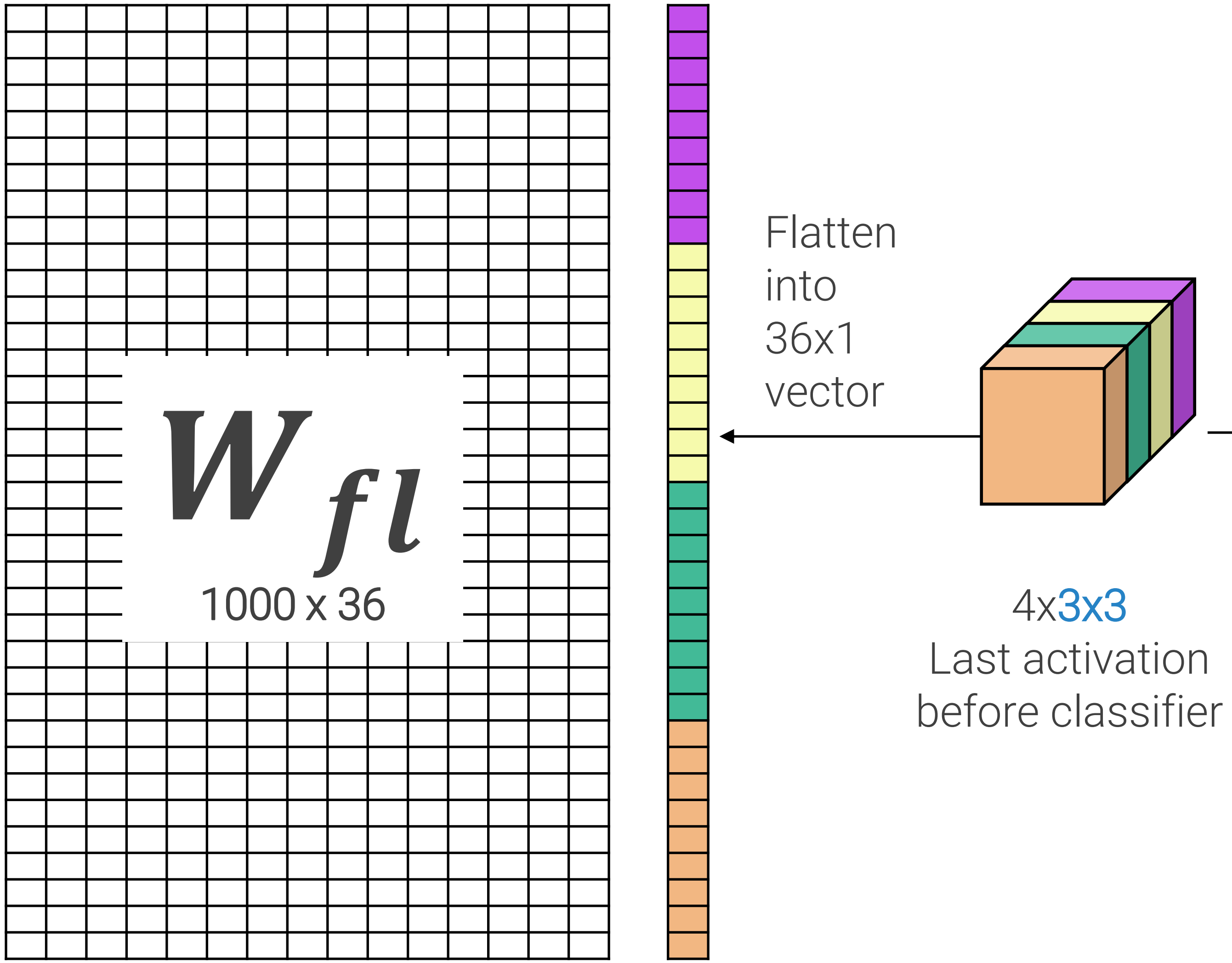
By adding 1x1 convolutions before larger convs and after max pool we can

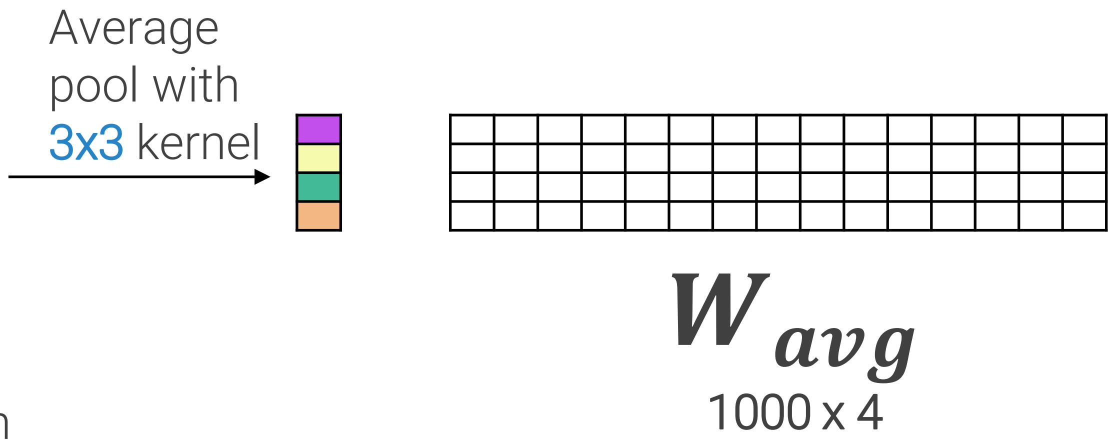
- Control the number of output channels by reducing the depth of the max pool output
- Control the time complexity of the larger convolutions by reducing the channel dimension, e.g. now flops are

Conv 5x5 = 28x28x16x192x2 + 28x28x32x16x5x5x2≅5M+10M=**25M flops (was 240M)**

Conv 3x3=28x28x96x192x2 + 28x28x128x96x3x3x2 ≅29M+173M=**202M flops (was 350M)**

Fully-connected classifier vs global average pooling





To reduce the number of parameters needed at the interface between convolutional features and fully connected layers, NiN proposed to get rid of spatial dimensions by averaging them out

Lin et al., "Network in network", ICLR 2014

GoogLeNet: Global Average Pooling

	inception 1x1				inception 3x3			Inception 5x5			Maxpool		Activations						
Layer	Ks	H/W	S	P	C_out	1x1	H/W	C_out	1x1	H/W	1x1	H/W	H/W	Channels	#Activations	#params (K)	flops (M)	Acts memory (MB)	Params memory (MB)
incep8	384	1	1	0	384	192	3	128	48	5	128	3	7	1024	50176	1443	16689,7	49,0	16,5
avgpool	1	7	1	0									1	1024	1024	0	6,4	1,0	0,0
fc1	1000	1	1	0									1	1000	1000	1025	262,1	1,0	11,7

GoogLeNet uses global average pooling to remove spatial dimensions and one FC layer to produce class scores.

- It results in 1 million parameters and negligible numbers of flops

VGG has 124 millions parameters in the final 3 fc layers, which requires 31 Gflops to compute.

- If the kernel size of pooling covering the full input activation is computed by the layer instead of being specified by the user, i.e. `AdaptiveAvgPool2d` layer instead of `AvgPool2d` in PyTorch , this makes the network able (at least as far as tensor dimensions are concerned) to work on any input image size.

CLASS `torch.nn.AdaptiveAvgPool2d(output_size: Union[T, Tuple[T, ...]])` [SOURCE]

Applies a 2D adaptive average pooling over an input signal composed of several input planes.

GoogLeNet summary

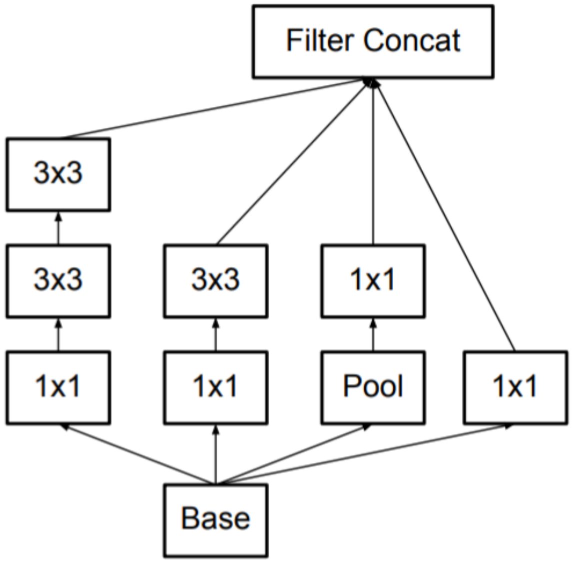
Full of magi numers = arbitraty decisions

	inception 1x1				inception 3x3			Inception 5x5			Maxpool		Activations					Acts memory	Params memory
Layer	Ks	H/W	S	P	C_out	1x1	H/W	C_out	1x1	H/W	1x1	H/W	H/W	Channels	#Activations	#params (K)	flops (M)	(MB)	(MB)
input													224	3	150528	0	-	73,5	0,0
conv1	64	7	2	3									112	64	802816	9	30211,6	784,0	0,1
pool1	1	3	2	1									56	64	200704	0	231,2	196,0	0,0
conv2	64	1	1	0									56	64	200704	4	3288,3	196,0	0,0
conv3	192	3	1	1									56	192	602112	111	88785,0	588,0	1,3
pool2	1	3	2	1									28	192	150528	0	173,4	147,0	0,0
incep1	64	1	1	0	128	96	3	32	16	5	32	3	28	256	200704	163	31380,5	196,0	1,9
incep2	128	1	1	0	192	128	3	96	32	5	64	3	28	480	376320	388	75683,1	367,5	4,4
pool3	1	3	2	1									14	480	94080	0	108,4	91,9	0,0
incep3	192	1	1	0	208	96	3	48	16	5	64	3	14	512	100352	376	17403,4	98,0	4,3
incep4	160	1	1	0	224	112	3	64	24	5	64	3	14	512	100352	449	20577,8	98,0	5,1
incep5	128	1	1	0	256	128	3	64	24	5	64	3	14	512	100352	509	23609,2	98,0	5,8
incep5	112	1	1	0	288	144	3	64	32	5	64	3	14	528	103488	605	28233,4	101,1	6,9
incep6	256	1	1	0	320	160	3	128	32	5	128	3	14	832	163072	867	41445,4	159,3	9,9
pool4	1	3	2	1									7	832	40768	0	47,0	39,8	0,0
incep7	256	1	1	0	320	160	3	128	32	5	128	3	7	832	40768	1042	11860,0	39,8	11,9
incep8	384	1	1	0	384	192	3	128	48	5	128	3	7	1024	50176	1443	16689,7	49,0	16,5
avgpool	1	1	1	0									1	1024	1024	0	6,4	1,0	0,0
fc1	1000	1	1	0									1	1000	1000	1025	262,1	1,0	11,7
											Minibatch:			128	Totals:	6.992	389.996	3.251	80

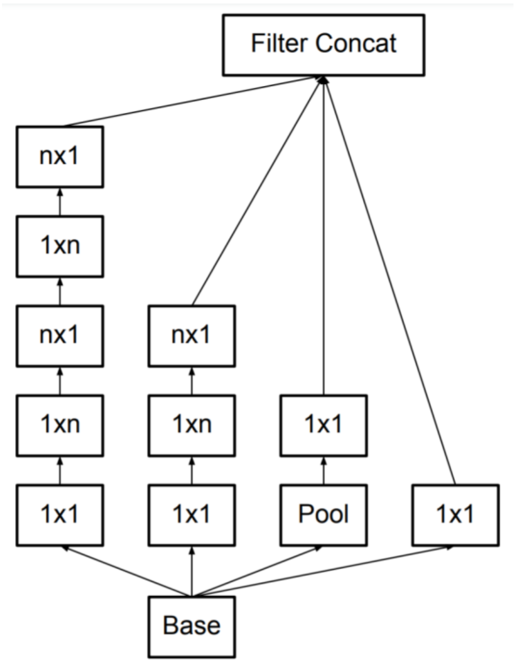
Only 7 millions parameters, 390 Gflops (3 Gflops/img), and 3.3 GB of memory. 10.07% error rate on ILSVRC 14 validation set with one model and one test crop, 7.89 with one model and aggressive cropping (144 crops)

Inception v3

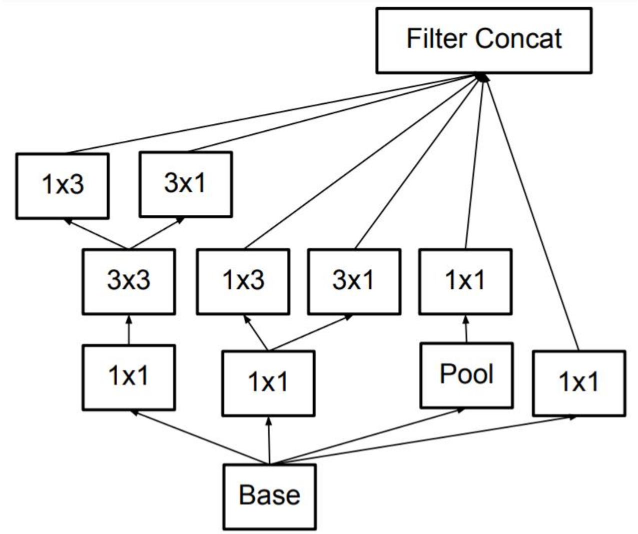
It leverages convolution factorizations to increase computational efficiency and to reduce the number of parameters. Less parameters should be more disentangled and therefore easier to train. Achieved 4.48% top-5 error on ILSVRC12 with 12 crops.



Factorization A, used for fine-scale activations (35x35)



Factorization B, used for mid-scale activations (17x17)



Factorization C, used for coarse-scale activations (8x8)

C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. "Rethinking the inception architecture for computer vision", CVPR 2016

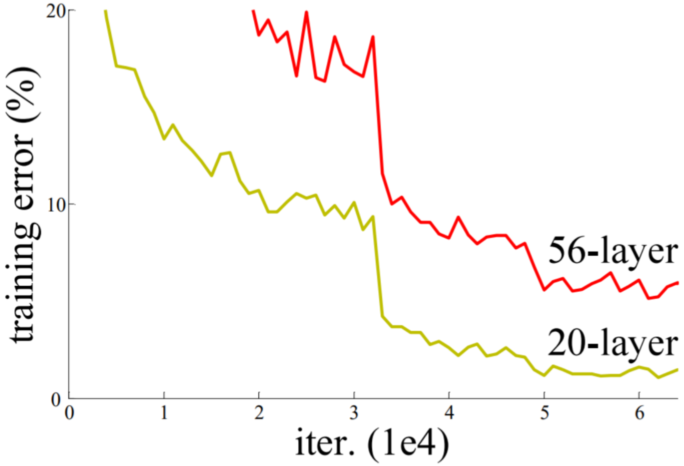
Residual Networks

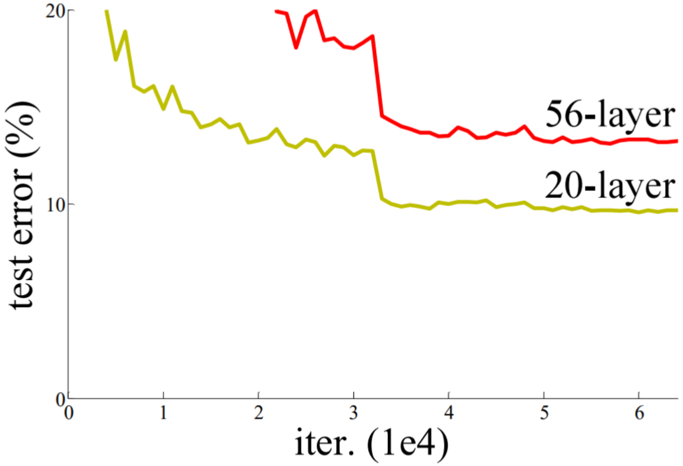
VGG lesson: growing depth improves performance. Yet, stacking more layers doesn't automatically improve performance.

Too many parameters increase overfitting and hurts generalization? We also observe higher training errors, so overfitting it's not the only reason, there is also a training problem, even when using Batch Norm.

Yet, a solution exists by construction: if a network with 20 layers achieves performance X, then we can stack 36 more identity layers and we should keep performance at X.

SGD is not able to find this solution with the parameterization we use for layers: optimizing very deep networks is hard.



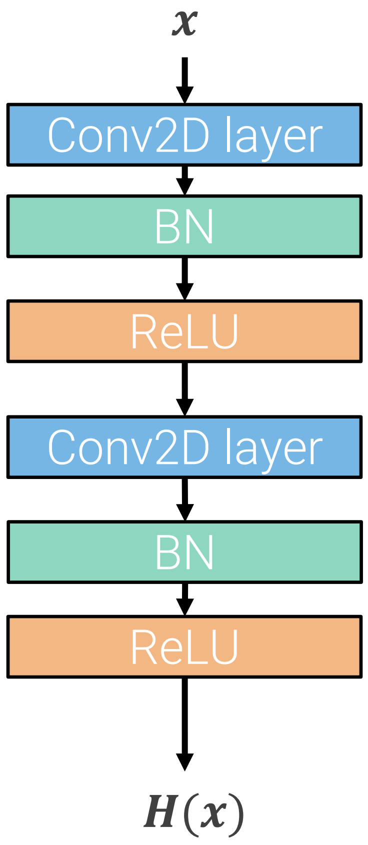


Kaiming He et al., "Deep Residual learning for image recognition", CVPR 2016

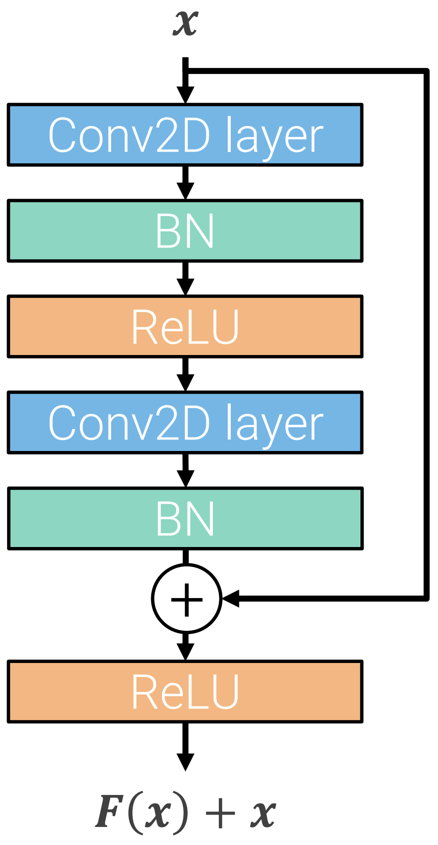
Residual block

Since SGD can't find the rigth solution (taht we know), let's start clorser to the solution

The proposed solution is to change the network so that learning identity functions is easy by introducing residual blocks. Implemented by adding skip connections skipping two convolutional layers.



<— SGD from this points —>



Weights usually initialized to be very small (or 0 for biases). Network starts with the identity function and learns an "optimal" perturbation of it.

It makes heavy use of batch-norm

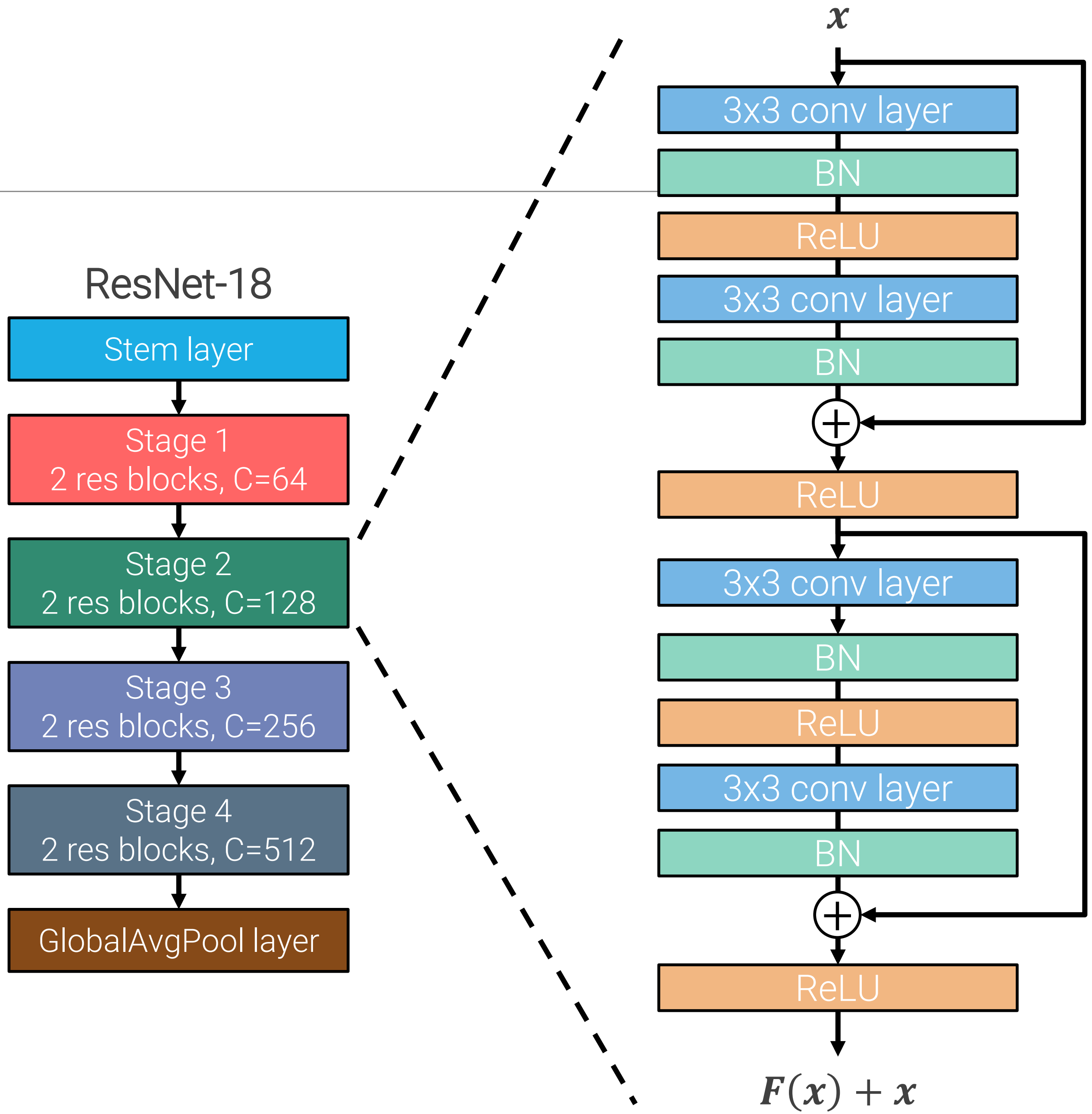
Residual Networks

Inspired by VGG regular design. Network is a stack of stages with fixed design rules:

- Stages are a stack of residual blocks
- Each residual block is a stack of two 3x3 convolutions with batch-norm
- the first block of each stage halves the spatial resolution (with stride-2 convs) and doubles the number of channels

It uses stem layer and global average pooling as GoogleLeNet

Naming conventions follow VGG, as well: ResNet-*X*, where *X* is the number of layers with learnable parameters



Stem layer and global average pooling

First layers are stem layers, as in GoogLeNet. However, it only uses one conv+pool layer, and reduces only to 56×56, probably because residual blocks are lightweight compared to Inception modules.

Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					224	3	150528	0	-	73,5	0,0
conv1	64	7	2	3	112	64	802816	9	30211,6	784,0	0,1
pool1	1	3	2	1	56	64	200704	0	231,2	196,0	0,0

...

Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
conv.s4.b3.2	512	3	1	1	7	512	25088	2360	29595,0	24,5	27,0
avgpool	1	7	1	0	1	512	512	0	3,2	0,5	0,0
fc1	1000	1	1	0	1	1000	1000	513	131,1	1,0	5,9

It also uses the same average pooling + linear layer at the end.

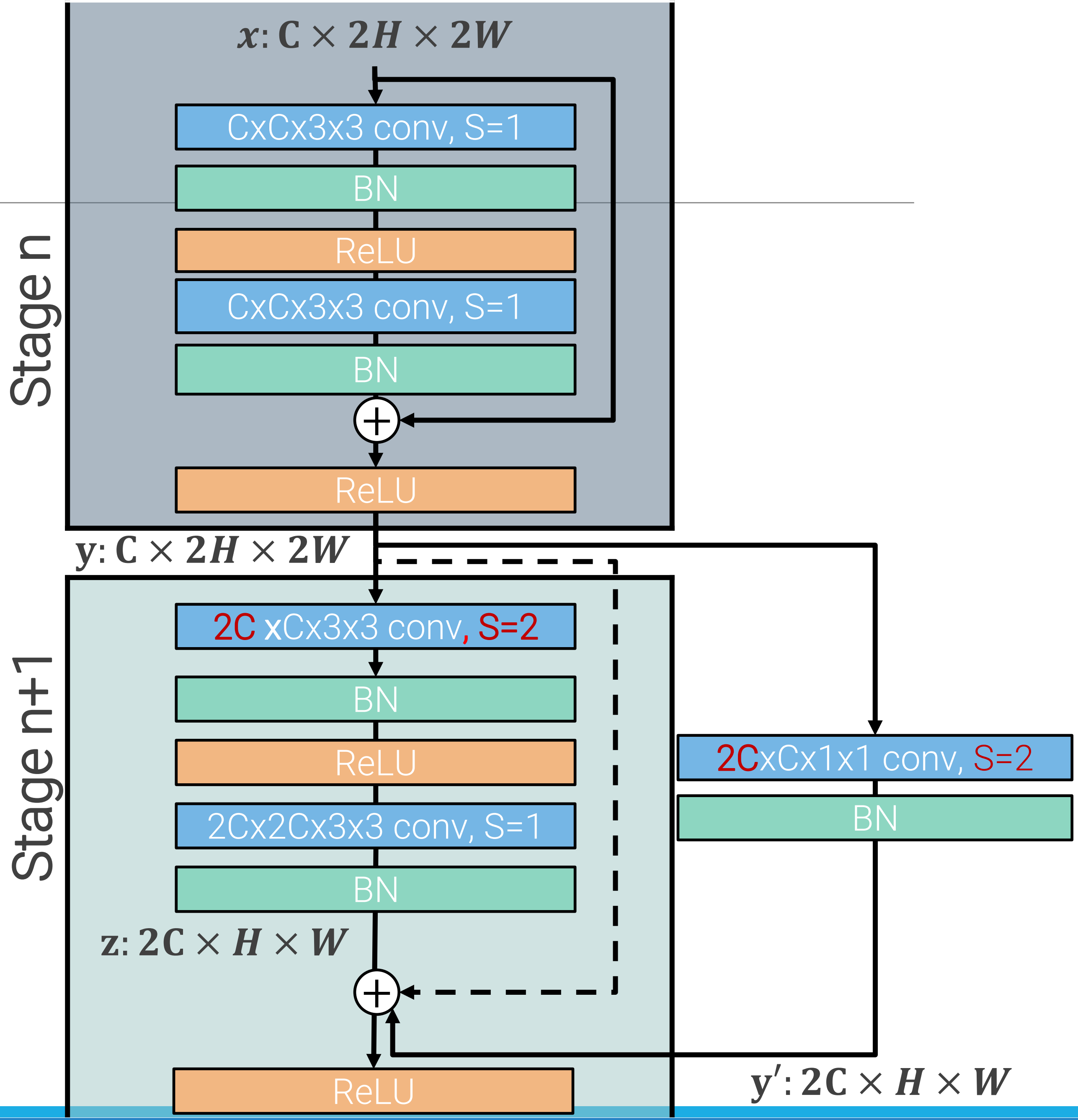
Skip conn dimensions

The residual blocks described so far **cannot be used as the first block of a new stage**, because the number of channels and the spatial dimensions do not match along the residual connection (dashed arrow)

The authors tried two solutions:

- apply stride 2 to y and zero pad the missing channels (no extra parameters)
- **1x1 conv with stride 2 and 2C output channels** (solid arrow)

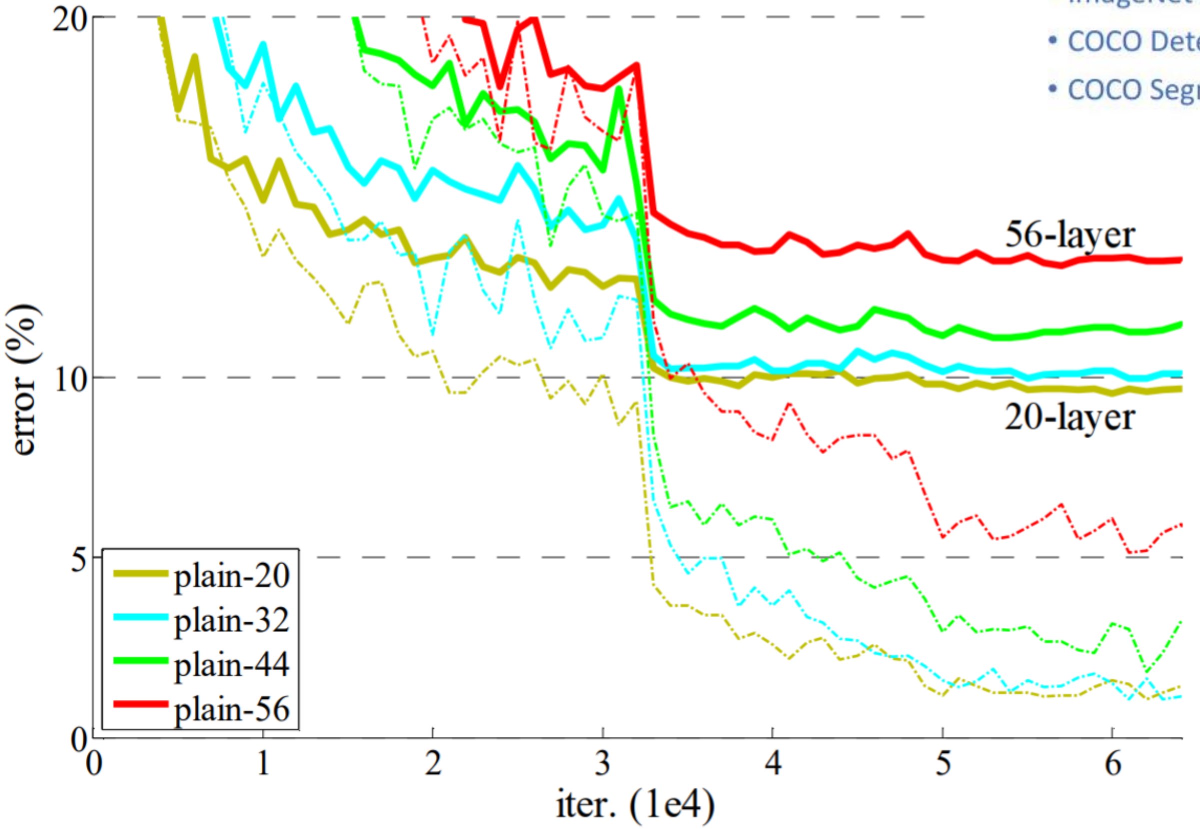
and verified the second one to perform slightly better

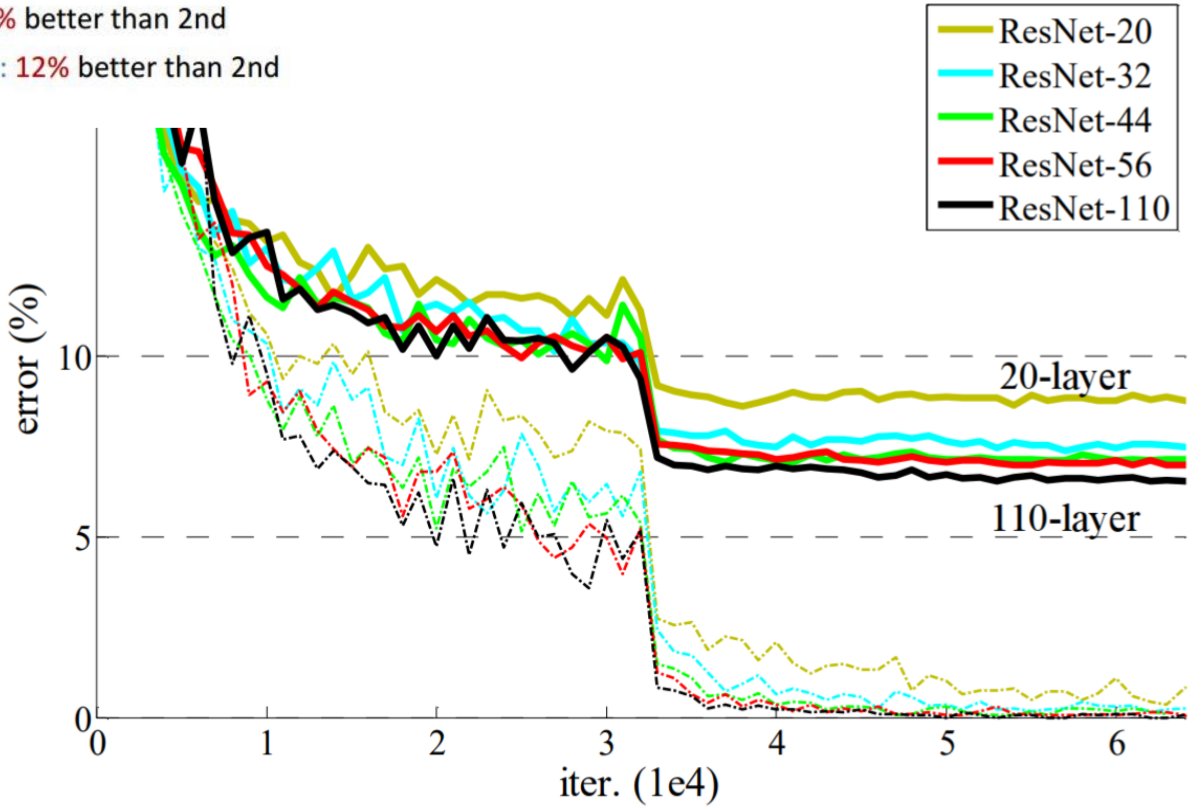


Results updated

MSRA @ ILSVRC & COCO 2015 Competitions

- **1st places in all five main tracks**
  - ImageNet Classification: *"Ultra-deep"* (quote Yann) 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

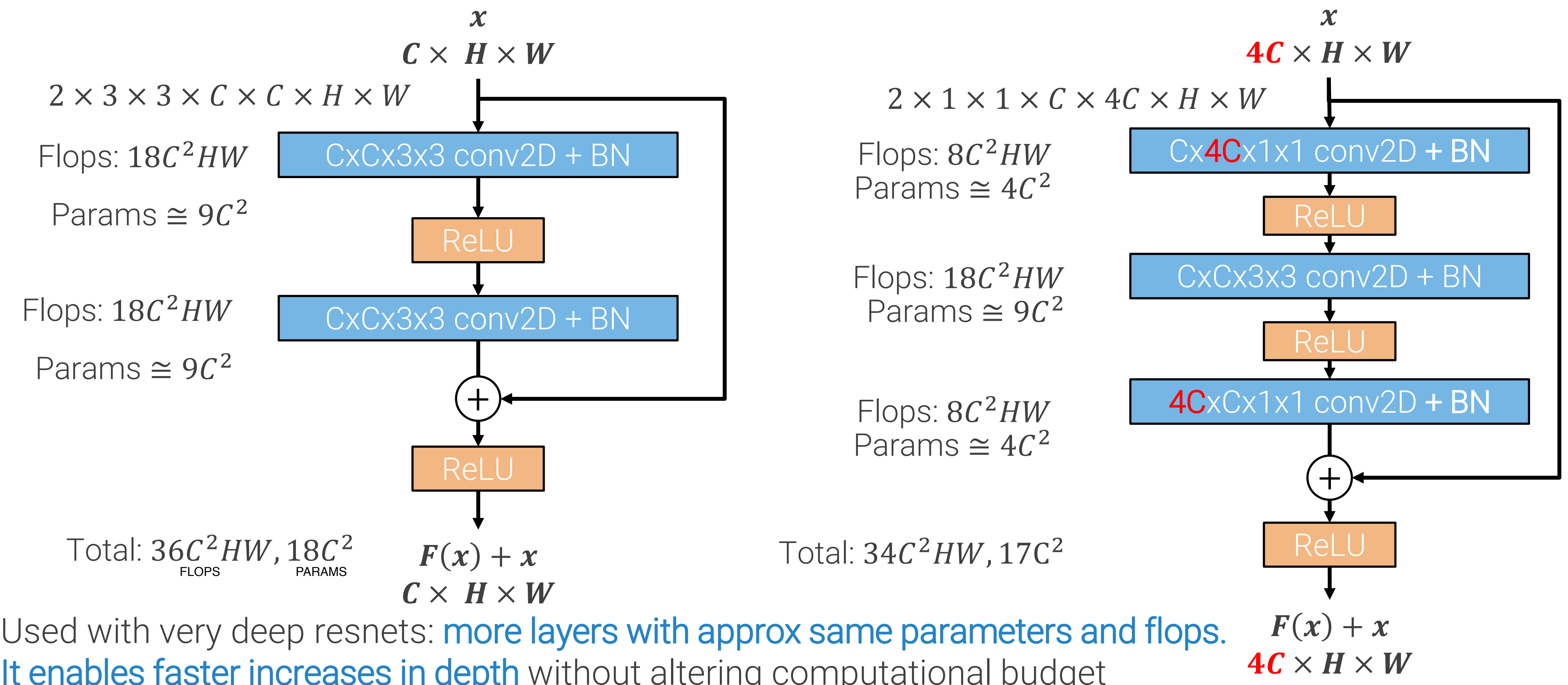




Residual blocks allow us to train deep networks. When properly trained, deep networks outperform shallower network as expected

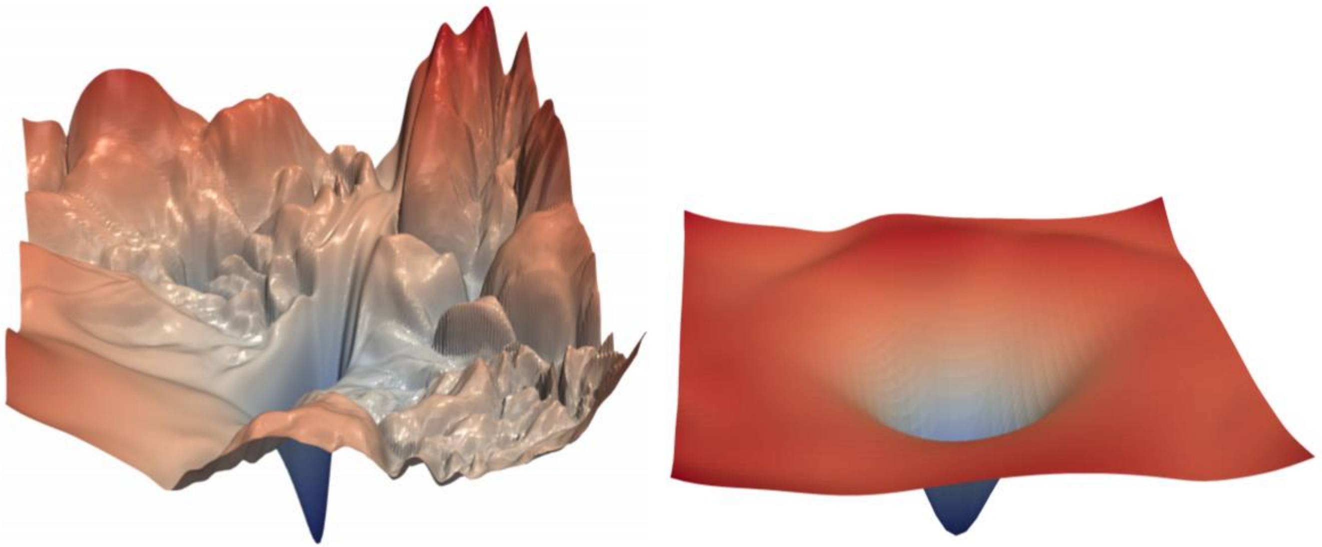
Won all 2015 competitions by a large margin, still the standard baseline/backbone for most tasks today.

Bottleneck residual block



Used with very deep resnets: more layers with approx same parameters and flops.
It enables faster increases in depth without altering computational budget

Effects of residual learning



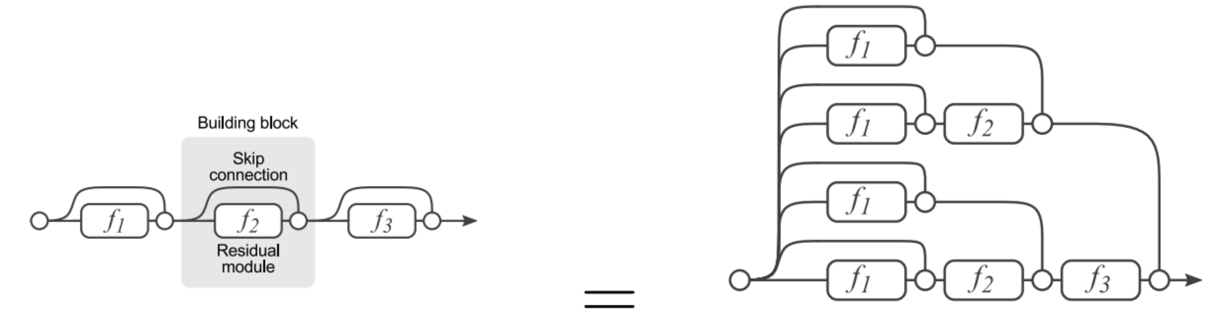
(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

Easier to find the minimum fors SGD in the 2nd space rathe rthan in the 1st

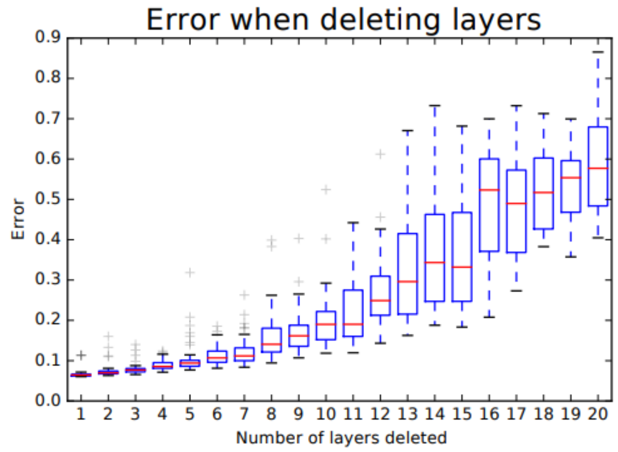
Hao Li et al., "Visualizing the Loss Landscape of Neural Nets", NeurIPS 2018

Resnets as ensembles of relatively shallow networks

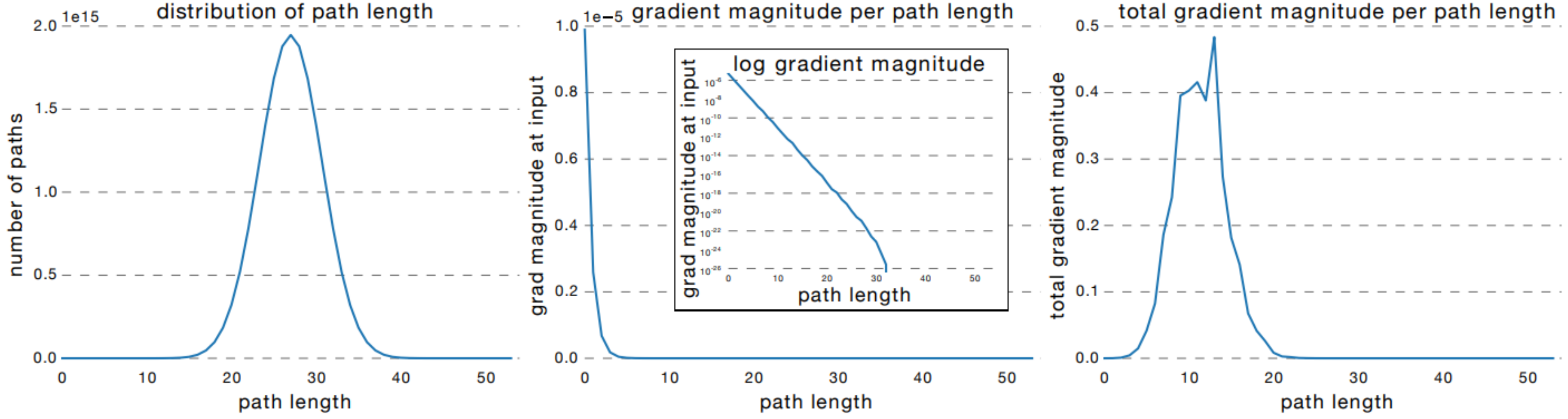


(a) Conventional 3-block residual network

(b) Unraveled view of (a)



The higher the number of layers detected and the higher the error

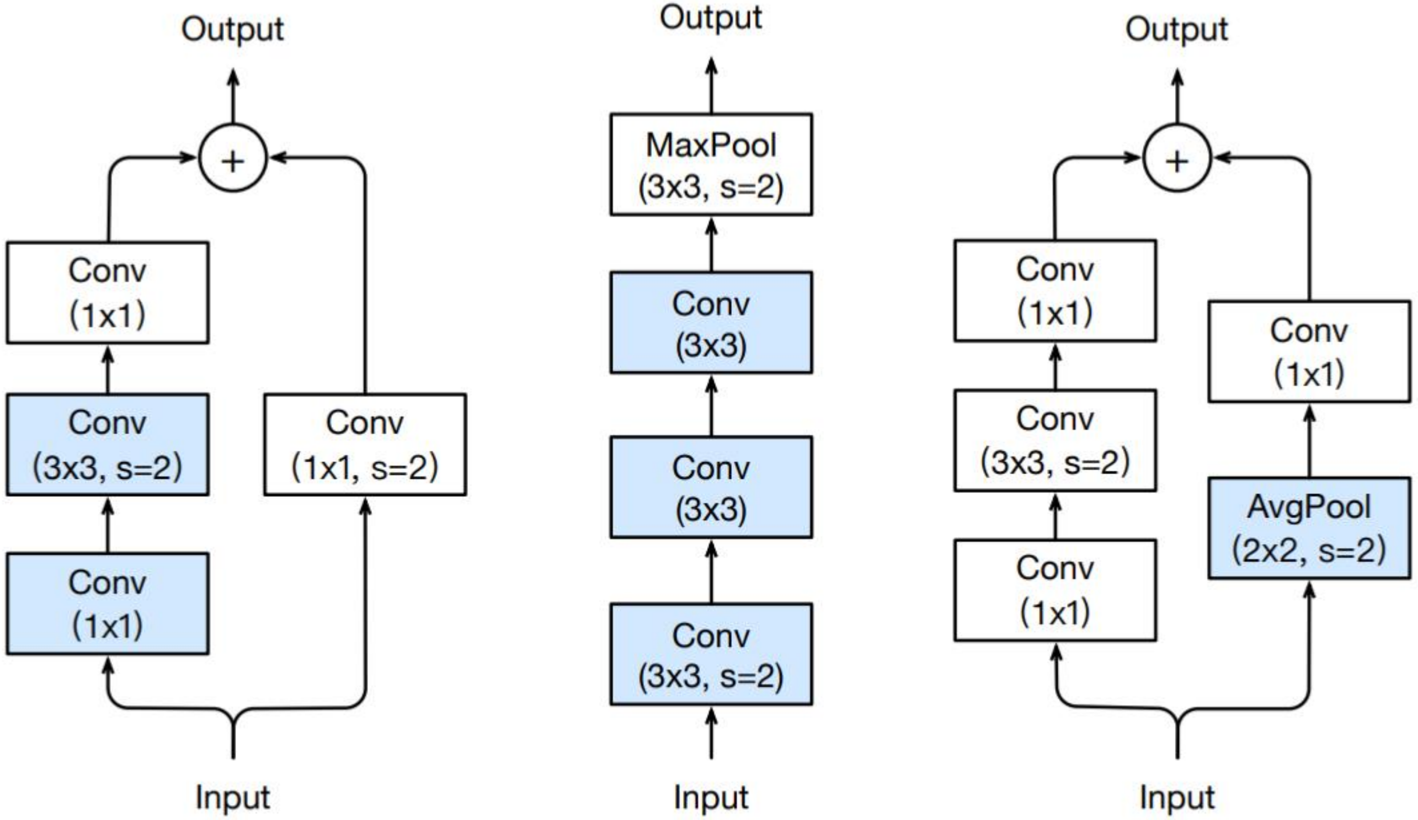


(a) (b) (c)

Andreas Veit, Michael Wilber, Serge Belongie. “Residual Networks Behave Like Ensembles of Relatively Shallow Networks”, NeurIPS 2016

Further model tweaks – "ResNet v2"

- ResNet-B: 1x1 convolution at the beginning of bottleneck residual blocks ignore ¾ of the input activation. If we move stride 2 into the 3x3, all input is used.
- ResNet-C: replace 7x7 stride 2 conv in stem layers with 3 3x3 convs, the first one with stride 2
- ResNet-D: the 1x1 stride 2 conv used to match dimensions in the first block of each stage uses only ¾ of the input activation. Performing 2x2 stride 2 AvgPool before it fixes the problem.



(a) ResNet-B (b) ResNet-C (c) ResNet-D

Tong He et al., "Bag of Tricks for Image Classification with Convolutional Neural Networks", CVPR 19

Common variants on ImageNet

ResNet-18	ResNet-34	ResNet-50	ResNet-101	ResNet-152
Stem layer	Stem layer	Stem layer	Stem layer	Stem layer
Stage 1 2 res blocks, C=64	Stage 1 3 res blocks, C=64	Bottleneck Stage 1 3 res blocks, C=256	Bottleneck Stage 1 3 res blocks, C=256	Bottleneck Stage 1 3 res blocks, C=256
Stage 2 2 res blocks, C=128	Stage 2 4 res blocks, C=128	Bottleneck Stage 2 4 res blocks, C=512	Bottleneck Stage 2 4 res blocks, C=512	Bottleneck Stage 2 8 res blocks, C=512
Stage 3 2 res blocks, C=256	Stage 3 6 res blocks, C=256	Bottleneck Stage 3 6 res blocks, C=1024	Bottleneck Stage 3 23 res blocks, C=1024	Bottleneck Stage 3 36 res blocks, C=1024
Stage 4 2 res blocks, C=512	Stage 4 3 res blocks, C=512	Bottleneck Stage 4 3 res blocks, C=2048	Bottleneck Stage 4 3 res blocks, C=2048	Bottleneck Stage 4 3 res blocks, C=2048
GlobalAvgPool+FC	GlobalAvgPool+FC	GlobalAvgPool+FC	GlobalAvgPool+FC	GlobalAvgPool+FC

Use standard blocks

Use bottleneck blocks, 4 times #channels

Common variants on ImageNet

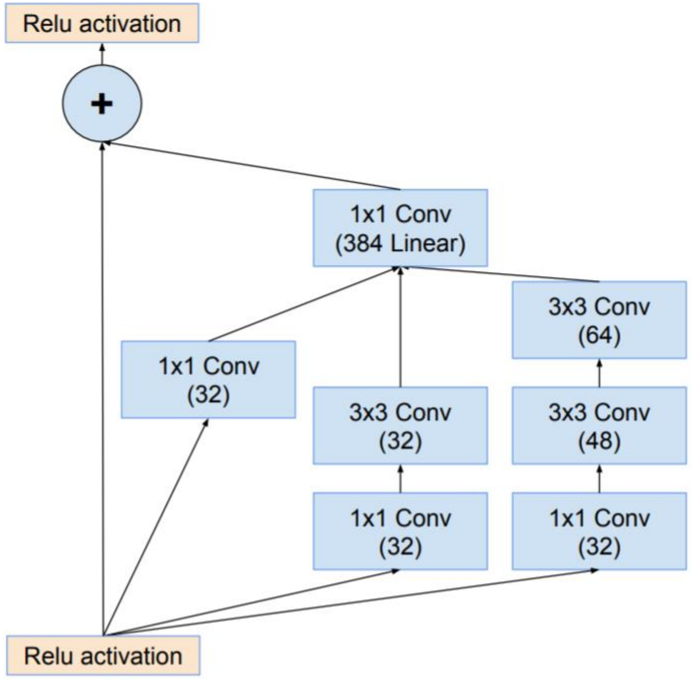
Good default choice

ResNet-18	ResNet-34	ResNet-50	ResNet-101	ResNet-152
Stem layer	Stem layer	Stem layer	Stem layer	Stem layer
Stage 1 2 res blocks, C=64	Stage 1 3 res blocks, C=64	Bottleneck Stage 1 3 res blocks, C=256	Bottleneck Stage 1 3 res blocks, C=256	Bottleneck Stage 1 3 res blocks, C=256
Stage 2 2 res blocks, C=128	Stage 2 4 res blocks, C=128	Bottleneck Stage 2 4 res blocks, C=512	Bottleneck Stage 2 4 res blocks, C=512	Bottleneck Stage 2 8 res blocks, C=512
Stage 3 2 res blocks, C=256	Stage 3 6 res blocks, C=256	Bottleneck Stage 3 6 res blocks, C=1024	Bottleneck Stage 3 23 res blocks, C=1024	Bottleneck Stage 3 36 res blocks, C=1024
Stage 4 2 res blocks, C=512	Stage 4 3 res blocks, C=512	Bottleneck Stage 4 3 res blocks, C=2048	Bottleneck Stage 4 3 res blocks, C=2048	Bottleneck Stage 4 3 res blocks, C=2048
GlobalAvgPool+FC	GlobalAvgPool+FC	GlobalAvgPool+FC	GlobalAvgPool+FC	GlobalAvgPool+FC
Errors are on the validation set with 10 crops	~7.3 Gflops/img ~22M params 7.46 top-5 error	~7.7 Gflops/img ~25M params 6.71 top-5 error	~15.1 Gflops/img ~44M params 6.05 top-5 error	~23.4 Gflops/img ~62M params 5.71 top-5 error

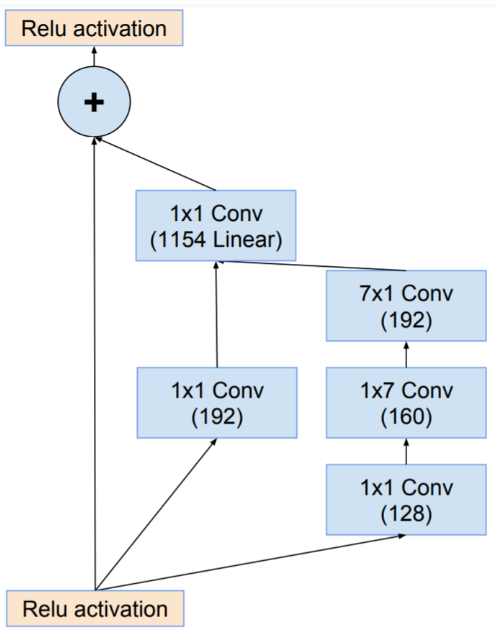
Inception-v4 and Inception-ResNet-v2

Inception-v4 is basically a larger Inception-v3 with a more complicated stem.

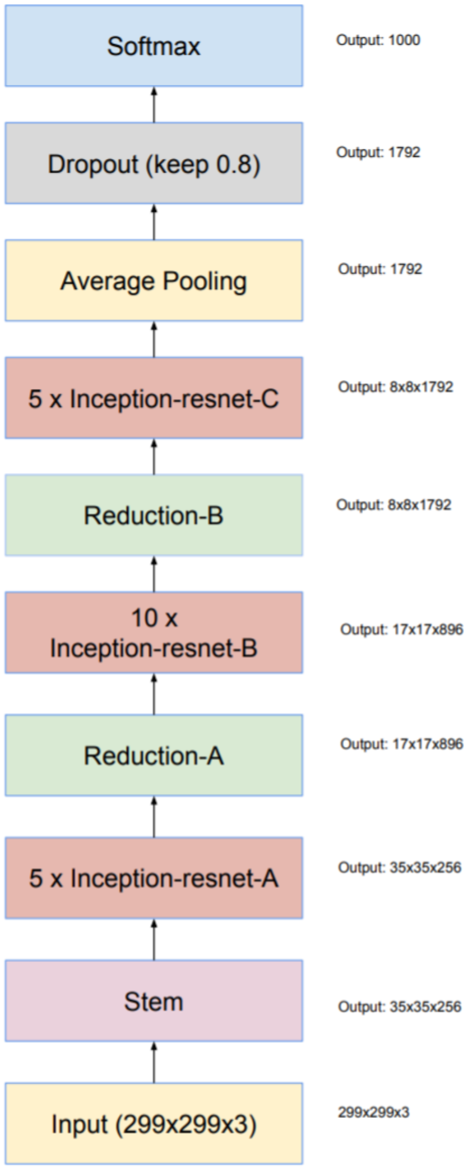
The authors also tried the residual connections idea around the Inception module.



Inception-resnet A, used for fine-scale activations (35x35)



Inception-resnet B, used for mid-scale activations (17x17)

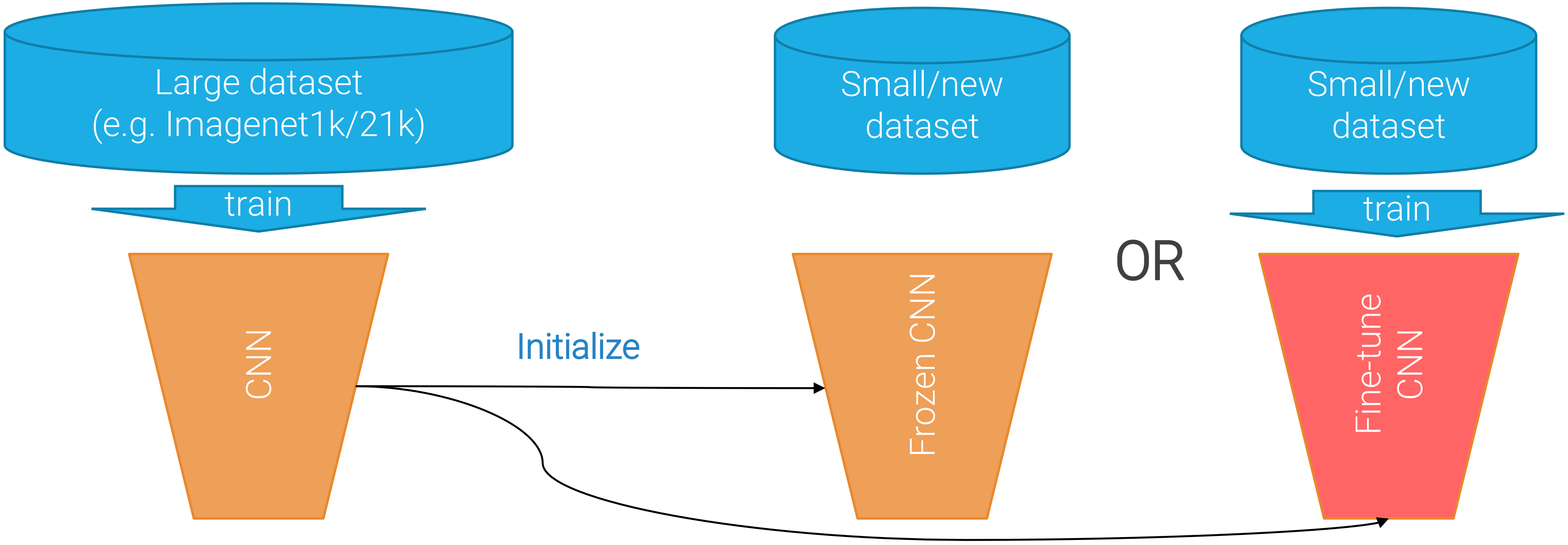


Christian Szegedy et al., "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning", AAAI 2017

Transfer Learning

We normally want to run CNNs on new classification datasets, not on ImageNet.

One of the most important features, from a practical point of view, of learned representations is that they can be effectively transferred to new datasets. Transfer learning is the process of using and adapting a pre-trained NN to new datasets. Usually, we pre-train on large datasets, and then we use it as frozen feature extractor or fine-tune it on the new dataset.



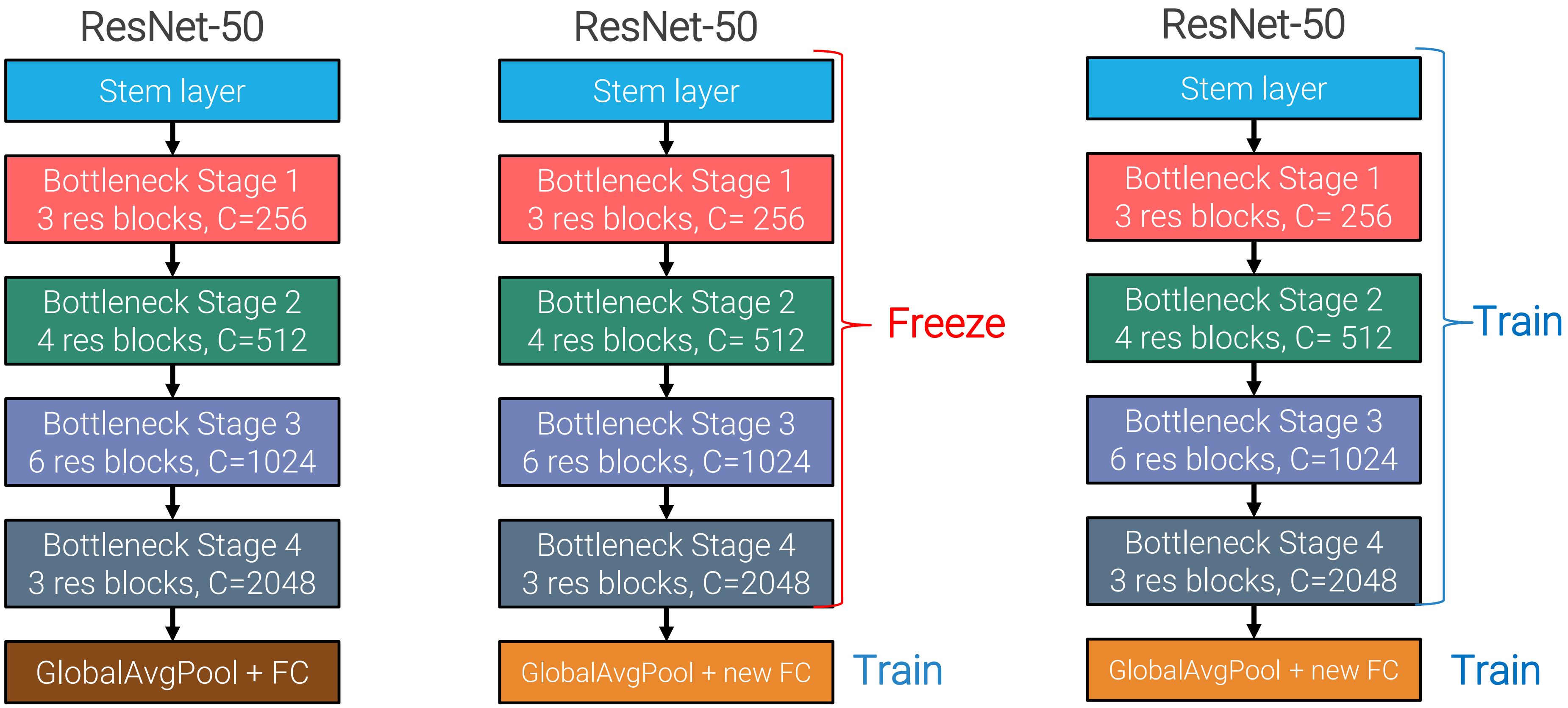
Transfer learning

1. Trained on Imagenet (find parameters online)

2a. Use as **pre-trained feature extractor**

OR

2b. **Fine-tuning**: train new head and feature extractor



When fine-tuning:

1. Start with frozen feature extractor
2. Use smaller LR than the one used to train original architecture
3. Progressive LRs: use smaller learning rates when training extractor, and even freeze lower layers